O'REILLY*

Head First **Ruby** A Brain-Friendly Guide



Jay McGavren

Head First Ruby

Jay McGavren



Beijing • Cambridge • Köln • Sebastopol • Tokyo

Head First Ruby

by Jay McGavren

Copyright © 2015 Jay McGavren. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/ institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editors:	Meghan Blanchette, Courtney Nash
Cover Designer:	Randy Comer
Production Editor:	
Indexer:	
Proofreader:	
Page Viewer:	

Printing History:

April 2015: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. The *Head First* series designations, *Head First Ruby*, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-37265-1 [LSI]

1 more with less

Code the Way You Want



You're wondering what this crazy Ruby language is all about,

and if it's right for you. Let us ask you this: *Do you like being productive?* Do you feel like all those extra compilers and libraries and class files and keystrokes in your other language bring you closer to a **finished product**, **admiring co-workers**, and **happy customers**? Would you like a language that **takes care of the details** for you? If you sometimes wish you could stop maintaining boilerplate code and *get to work on your problem*, then Ruby is for you. Ruby lets you **get more done with less code**.



The Ruby Philosophy

Back in the 1990's in Japan, a programmer named Yukihiro Matsumoto ("Matz" for short) was dreaming about his ideal programming language. He wanted something that:

- Was easy to learn and use
- Was flexible enough to handle any programming task
- Let the programmer concentrate on the problem they were trying to solve
- Gave the programmer less stress
- Was object-oriented

He looked at the languages that were available, but felt that none of them was exactly what he wanted. So, he set out to make his own. He called it Ruby.

After tinkering around with Ruby for his own work for a while, Matz released it to the public in 1995. Since then, the Ruby community has done some amazing things:

- Built out a vast collection of Ruby libraries that can help you do anything from reading CSV files to controlling objects over a network
- Written alternate interpreters that can run your Ruby code faster or integrate it with other languages
- Created Ruby on Rails, a hugely popular framework for web applications

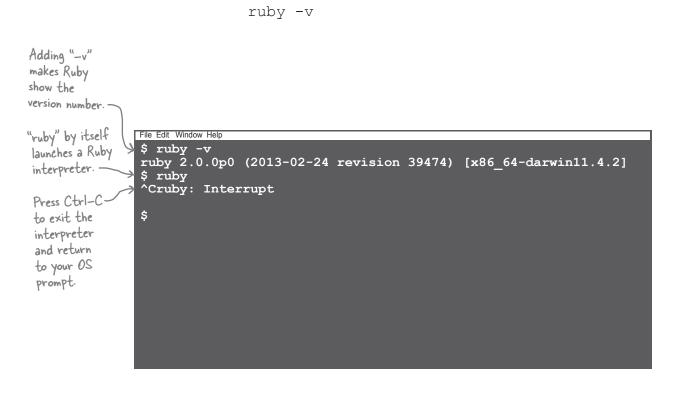
This explosion of creativity and productivity was enabled by the Ruby language itself. Flexibility and ease of use are core principles of the language, meaning you can use Ruby to accomplish any programming task, in fewer lines of code than other languages.

Once you've got the basics down, you'll agree: Ruby is a joy to use!

Flexibility and ease of use are core principles of Ruby.

Get Ruby

First things first: you can *write* Ruby code all day, but it won't do you much good if you can't *run* it. Let's make sure you have a working Ruby **interpreter** installed. We want version 2.0 or later. Open up a command-line prompt and type:



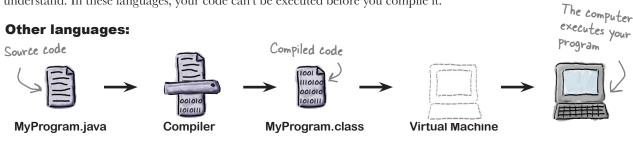
When you type **ruby -v** at a prompt, if you see a response like this, you're in business:

Use Ruby

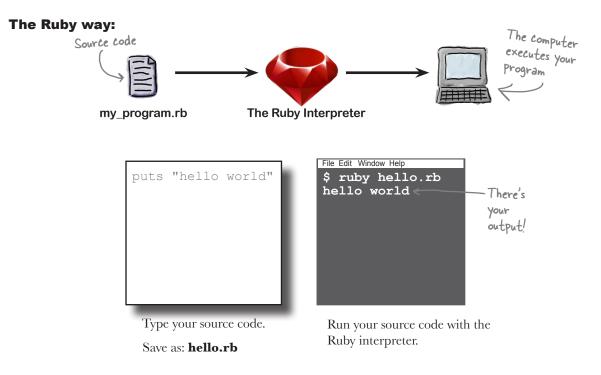
To run a Ruby script, you simply save your Ruby code in a file, and run that file with the Ruby interpreter. Ruby source files that you can execute are referred to as scripts, but they're really just plain text files.

You may be used to other languages (like C++, C#, or Java) where you have to manually compile your code to a binary format that a CPU or virtual machine can understand. In these languages, your code can't be executed before you compile it.





With Ruby, you skip that step. Ruby instantly and automatically compiles the source code in your script. This means less time between writing your code and trying it out!



Use Ruby - interactively

There's another big benefit to using a language like Ruby. Not only do you not have to run a compiler each time you want to try out your code, you don't even have to put it in a script first.

Ruby comes with a separate program, called irb (for Interactive **Rub**y). The irb shell lets you type any Ruby expression, which it will then immediately evaluate and show you the results. It's a great way to learn the language, because you get immediate feedback. But even Ruby professionals use irb to try out new ideas.

Throughout the book, we'll be writing lots of scripts to be run via the Ruby interpreter. But anytime you're testing out a new concept, it's a great idea to launch irb and experiment a bit.

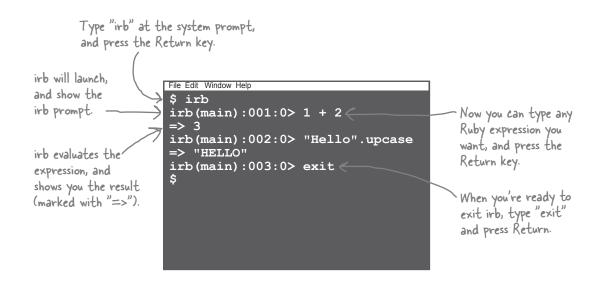
So what are we waiting for? Let's get into irb now and play around with some Ruby expressions.

Using the irb shell

Open a terminal window, and type irb. This will launch the interactive Ruby interpreter. (You'll know it's running because the prompt will change, although it may not match exactly what you see here.)

From there, you can type any expression you want, followed by the Return key. Ruby will instantly evaluate it and show you the result.

When you're done with irb, type exit at the prompt, and you'll be returned to your OS's system prompt.



Your first Ruby expressions

Now that we know how to launch irb, let's try a few expressions out and see what results we get!

Type the following at the prompt, then press Return:

1 + 2

You'll be shown the result:

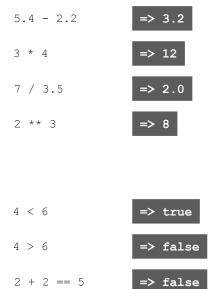
=> 3

If you type:

Math operations and comparisons

Ruby's basic math operators work just like they do in most other languages. The + symbol is for addtion, – for subtraction, * for multiplication, / for division, and ** for exponentiation. 3

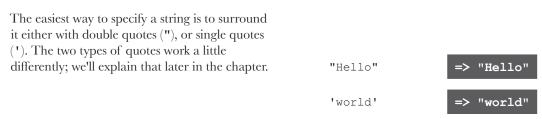
You can use < and > to compare two values and see if one is less than or greater than another. You can also use == (that's *two* equals signs) to see if two values are equal.



"irb" displays:

Strings

A **string** is a series of text characters. You can use them to hold names, e-mail addresses, phone numbers, and a million other things. Ruby's strings are special because even very large strings are highly efficient to work with (this isn't true in many other languages).



Variables

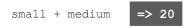
Ruby lets us create **variables** - names that refer to values.

You don't have to declare variables in Ruby; assigning to them creates them. You assign to a variable with the = symbol (that's a *single* equals sign).



A variable name starts with a lower-case letter, and can contain letters, numbers, and underscores.

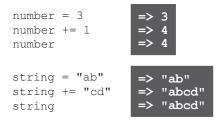
Once you've assigned to variables, you can access their values whenever you need, in any context where you might use the original value.

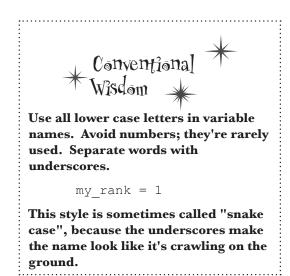


Variables don't have types in Ruby; they can hold any value you want. You can assign a string to a variable, then immediately assign a floating-point number to the same variable, and it's perfectly legal.



The += operator lets you add on to the existing value of a variable.





Everything is an object!

Ruby is an *object-oriented* language. That means your data has useful **methods** (fragments of code that you can execute on demand) attached directly to it.

In modern languages, it's pretty common for something like a string to be a full-fledged object, so of course strings have methods to call:

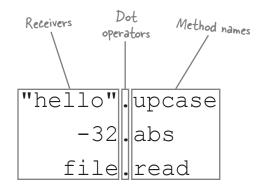


What's cool about Ruby, though, is that *everything* is an object. Even something as simple as a number is an object. That means they have useful methods, too.



Calling a method on an object

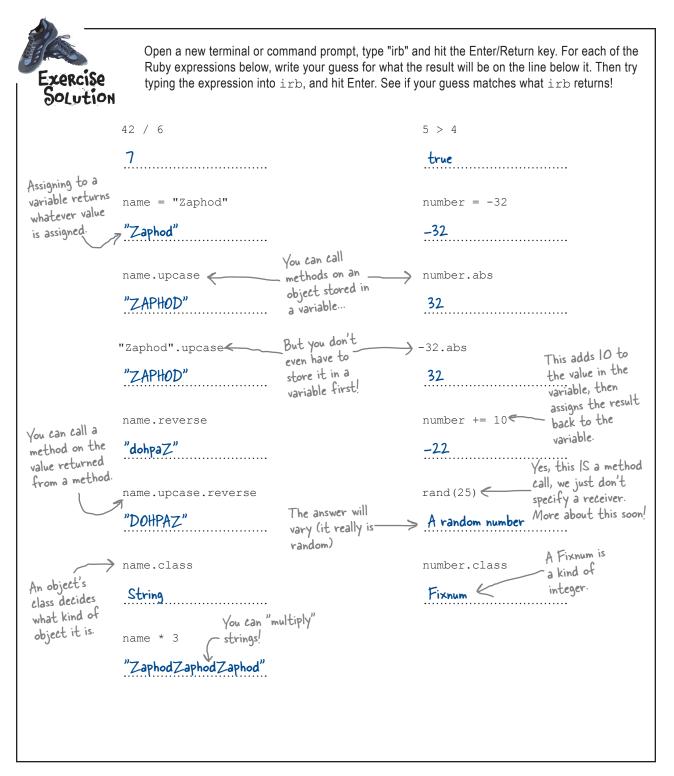
When you make a call like this, the object you're calling the method on is known as the method **receiver**. It's whatever is to the left of the dot operator. You can think of calling a method on an object like *passing it a message*. Like a note saying, "Hey, can you send me back an upper case version of yourself?" or "Can I have your absolute value?".





Open a new terminal or command prompt, type "irb" and hit the Enter/Return key. For each of the Ruby expressions below, write your guess for what the result will be on the line below it. Then try typing the expression into irb, and hit Enter. See if your guess matches what irb returns!

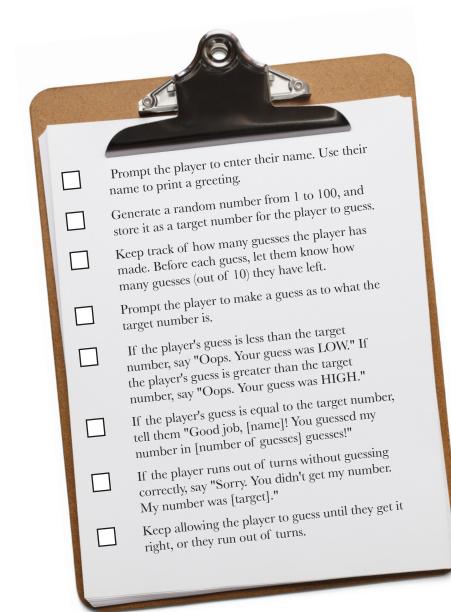
42 / 6	5 > 4
name = "Zaphod"	number = -32
name.upcase	number.abs
"Zaphod".upcase	-32.abs
name.reverse	number += 10
name.upcase.reverse	rand(25)
name.class	number.class
name * 3	



Let's build a game

In this first chapter, we're going to build a simple game. If that sounds daunting, don't worry; it's easy when you're using Ruby!

Let's look at what we'll need to do:

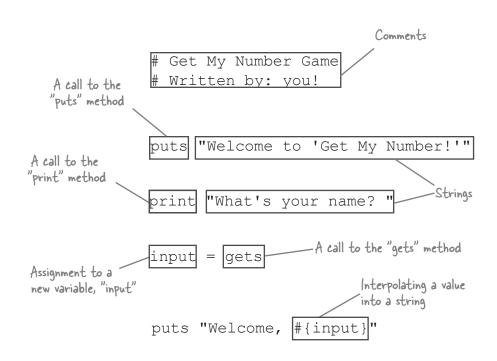




Input, storage, and output

Our first requirement is to greet the user by name. To accomplish that, we'll need to write a script that *gets input* from the user, *stores* that input, and then uses that stored value to create some *output*.

We can do all this in just a few lines of Ruby code:



We'll go into detail on each of the components of this script over the next few pages. But first, let's give it a try!

Running scripts

We've written a simple script that fulfills our first requirement: to greet the player by name. Now, we'll show you how to execute the script, so you can see what you've created.



Step One:

Open a new document in your favorite text editor, and type in the following code.

# Get My Number Game # Written by: you!	
puts "Welcome to 'Get My Number!'" print "What's your name? "	
input = gets	
puts "Welcome, #{input}"	
get_number.r	ь

Step Two:

Save the file as "get_number.rb".

Step Three:

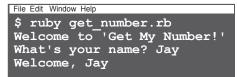
Open up a command-line prompt, and change into the directory where you saved your program.

Step Four:

Run the program by typing "ruby get_number.rb".

Step Five:

You'll see a greeting, and a prompt. Type your name and hit the Enter/Return key. You'll then see a message that welcomes you by name.



Let's take a few pages to look at each part of this code in more detail. Comments

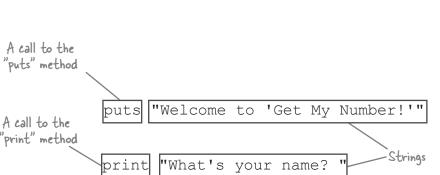
Our source file starts out with a couple comments. Ruby ignores everything from a hash mark (#) up until the end of the line, so that you can leave instructions or notes for yourself and your fellow developers.

If you place a pound sign (#) in your code, then everything from that point until the end of the line will be treated as a comment, and ignored by Ruby. This works just like the double-slash ("//") marker in Java or JavaScript.

```
i_am = "executed" # I'm not.
# Me neither.
```

"puts" and "print"

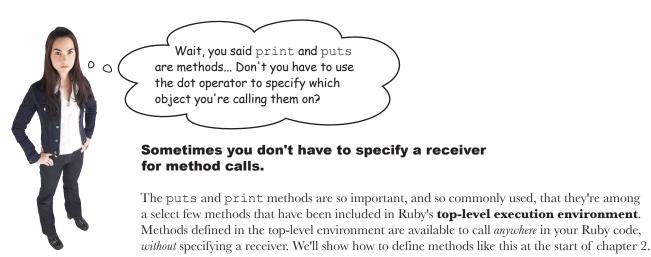
The actual code starts with a call to the puts method ("puts" is short for "<u>put string</u>"), which displays text on standard output (usually the terminal). We pass puts a string containing the text to display.



Get My Number Game

Written by: you!

We pass another string to the print method on the following line, to ask the user their name. The print method works just like puts, except that puts adds a newline character at the end of the string (if it doesn't already have one) to skip to the following line, whereas print doesn't. For cosmetic reasons, we end the string that we pass to print with a space, so that our text doesn't run up against the space where the user types their name.



Method arguments

The puts method takes a string and prints it to standard output (your terminal window).

puts "first line"

The string passed to the puts method is known as the method **argument**.

The puts method can take more than one argument; just separate the arguments with commas. Each argument gets printed on its own line.

puts "second line", "third line", "fourth line"

"gets"

The gets method (short for "get string") reads a line from standard input (characters typed in the terminal window). When you call it, it causes the program to halt until the user types their name and presses the Enter key. It returns the user's text to the program as another string.

Like puts and print, the gets method can be called from anywhere in your code without specifying a receiver.

Parenthesis are optional on method calls

Method arguments can be surrounded with parenthesis in Ruby:

puts("one", "two")

But the parenthesis are optional, and in the case of puts, most Rubyists prefer to leave them off.

puts "one", "two"

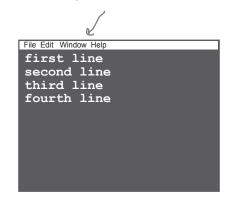
The gets method reads a line from standard input (characters typed in the terminal window). It doesn't (usually) need any arguments:

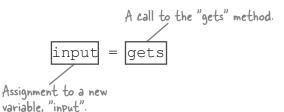
gets

Rubyists are *adamant* that parenthesis *not* be used if a method takes *no* arguments. So please, don't do this, even though it's valid code:

gets () Ko!

What it looks like in your terminal.



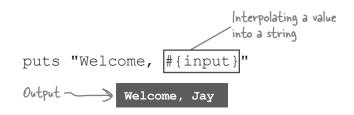




Leave parenthesis off of a method call if there are no arguments. You can leave them off for method calls where there *are* arguments as well, but this can make some code more difficult to read. When in doubt, use parenthesis!

String interpolation

The last thing our script does is to call puts with one more string. This one is special because we interpolate (substitute) the value in the name variable into the string. Whenever you include the # { . . . } notation *inside* a string, Ruby uses the value in the curly braces to "fill in the blank". The $\#\{\ldots\}$ markers can occur anywhere in the string: the beginning, end, or somewhere in the middle.



You're not limited to using variables within the # { } marker - you can use any Ruby expression.

puts "The answer is #{6 * 7}." Output ----> The answer is 42. Note that Ruby only applies interpolation in *double*-quoted strings. If you include a # { } marker in a *single*-quoted string, it will be taken literally. Output ----> Welcome, #{input}

puts 'Welcome, #{input}'

there are no Dumb Questions

Q: Where are the semicolons?

A: In Ruby, you *can* use semicolons to separate statements, but you generally shouldn't. (It's harder to read.)

> puts "Hello"; - No! puts "World";

Ruby treats separate lines as separate statements, making semicolons unnecessary.

> puts "Hello" puts "World"

Q: My other language would require me to put this script in a class with a "main" method. Doesn't Ruby?

A: No! That's one of the great things about Ruby - it doesn't require a bunch of ceremony for simple programs. Just write a few statements, and you're done!

Ruby doesn't require a bunch of ceremony for simple programs.

What's in that string?

File Edit Window Help

\$ ruby get_number.rb
Welcome to 'Get My Number!'
What's your name? Jay
Welcome, Jay



Well, that's easy enough to add. Let's throw an exclamation point on the end of the greeting string, after the interpolated value.

```
puts "Welcome to 'Get My Number!'"
print "What's your name? "
input = gets
puts "Welcome, #{input}!"
    Just this one little
    character added/
```

But if we try running the program again, we'll see that rather than appearing immediately after the user's name, the exclamation point jumps down to the next line!



Why is this happening? Maybe there's something going on within that input variable...

Printing it via the puts method doesn't reveal anything special about it, though:

puts input



Inspecting objects with the "inspect" and "p" methods

Now, let's try again, using a method meant especially for troubleshooting Ruby programs. The inspect method is available on any Ruby object. It converts the object to a string representation that's suitable for debugging. That is, it will reveal aspects of the object that don't normally show up in program output.

Here's the result of calling inspect on our string:

puts input.inspect "Jay\n" - Ah-HA!

What's that \n at the end of the string? We'll solve that mystery on the next page...

Printing the result of inspect is done so often that Ruby offers another shortcut: the p method. It works just like puts, except that it calls inspect on each argument before printing it.

This call to p is effectively identical to the previous code:

p input



Remember the p method; we'll be using it in later chapters to help debug Ruby code!

Escape sequences in strings

Our use of the p method has revealed some unexpected data at the end of the user's input:

p input



These two characters, the backslash character (\) and the n that follows it, actually represent a single character, a newline character. (The newline character is named thus because it makes terminal output jump down to a *new line*.) There's a newline at the end of the user input because when the user hits the Return key to indicate their entry is done, that gets recorded as an extra character. That newline is then included in the return value of the gets method.

The backslash character (\) and the n that follows it are an **escape sequence** - a portion of a string that represents characters that can't normally be represented in source code.

The most commonly-used escape sequences are n (newline, as we've seen), and t (a tab character, for indentation).

```
puts "First line\nSecond line\nThird line"
puts "\tIndented line"
```

First line
Second line
Third line
Indented line

Commonly-used escape sequences

lf you include this in a double-quoted string	you get this character
\n	newline
\t	tab
\"	double-quotes
\'	single-quote
	backslash

Normally, when you try to include a double-quotation mark (") in a double-quoted string, it gets treated as the end of the string, leading to errors:

If you escape the double-quotation marks by placing a backslash before each, you can place them in the middle of a double-quoted string.

```
puts "\"It's okay, \" he said."
```

"It's okay," he said.

Lastly, because $\ marks$ the start of an escape sequence, we also need a way to represent a backlash character that *isn't* part of an escape sequence. Using $\ will give us a literal backslash.$

puts "One backslash: \\"

One backslash: \

Bear in mind that most of these escape sequences apply only in *double*-quoted strings. In *single*-quoted strings, most escape sequences are treated literally.

 $n\t$ "

Calling "chomp" on the string object

File Edit Window Help \$ ruby get_number.rb Welcome to 'Get My Number!' What's your name? Jay Welcome, Jay

Okay, so the output is messed up because the user input string has a newline character at the end of it. What can we do about that?

0

We can use the chomp method to remove the newline character.

If the last character of a string is a newline, the chomp method will remove it. It's great for things like cleaning up strings returned from gets.

The chomp method is more specialized than print, puts, and gets, so it's available only on individual string objects. That means we need to specify that the string referenced by the input variable is the *receiver* of the chomp method. We need to use the dot operator on input.

	# Get My Number Game # Written by: you!
We'll store the return	puts "Welcome to 'Get My Number!'" print "What's your name? "
value of "chomp" in a new variable, "name".	input = gets Calling the "chomp" method.
The string in "input" -	puts "Welcome, #{name}!"
is the receiver of the "chomp" method.	We'll use "name" in the greeting, instead of "input".

The chomp method returns the same string, but without the newline character at the end. We store this in a new variable, name, which we then print as part of our welcome message.

If we try running the program again, we'll see that our new, emphatic greeting is working properly now!

File Edit Window Help \$ ruby get_number.rb Welcome to 'Get My Number!' What's your name? Jay Welcome, Jay!

What methods are available on an object?

You can't call just any method on just any object. If you try something like this, you'll get an error:

puts 42.upcase Error		se' for 42:Fixnum (NoMethodError)
Which, if you think about it, isn't s lot of sense to capitalize a number,	o wrong. After all, it doesn't make a does it?	
But, then, what methods <i>can</i> you ca question can be answered with a m		2. methods to s abs odd? many more to list here!
If you call methods on a string, you'll get a different list:		ngth case Plus too many

Why the difference? It has to do with the object's class. A **class** is a blueprint for making new objects, and it decides, among other things, what methods you can call on the object.

There's another method that lets objects tell us what their class is. It's called, sensibly enough, class. Let's try it out on a few objects.

```
puts 42.class Fixnum
puts "hello".class String
puts true.class TrueClass
```

We'll be talking more about classes in the next chapter, so stay tuned!



That's all the code for our first requirement. You can check it off the list!

Generating a random number

Our player greeting is done. Let's look at our next requirement.



The rand method will generate a random number within a given range. It should be able to create a target number for us.

We need to pass an argument to rand with the number that will be at the upper end of our range (100). Let's try it out a couple times:

67

25

puts rand(100) puts rand(100)

Looks good, but there's one problem: rand generates numbers between *zero* and *just below* the maximum value you specify. That means we'll be getting random numbers in the range 0-99, not 1-100 like we need.

Get My Number Game That's easy to fix, though, we'll just add 1 to whatever value we get # Written by: you! back from rand. That will put us puts "Welcome to 'Get My Number!'" back in the range of 1-100! # Get the player's name, and greet them. rand(100) + 1print "What's your name? " input = gets name = input.chomp puts "Welcome, #{name}!" We'll store the result in a new # Store a random number for the player to guess. variable, named target. puts "I've got a random number between 1 and 100." puts "Can you guess it?" code target = rand(100) + 1

Converting to strings

That's another requirement down! Let's look at the next one...

Generate a random number from 1 to 100, and store it as a target number for the player to guess.

Keep track of how many guesses the player has made. Before each guess, let them know how many guesses (out of 10) they have left.

"Keep track of how many guesses the player has made..." Looks like we'll need a variable for the number of guesses. Obviously, when the player first starts, they haven't made any guesses, so we'll create a variable named num guesses that's set to 0 initially.

num guesses = 0

Now, the first thing you might attempt to do in order to display the number of guesses remaining is to concatenate (join) the strings together using the + sign, as many other languages do. Something like this *won't work*, however:

The + operator is used to *add numbers* as well as to *concatenate strings*, and since remaining_guesses contains a number, this plus sign looks like an attempt to add numbers.

What's the solution? You need to convert the number to a string. Almost all Ruby objects have a to_s method you can call to do this conversion; let's try that now.

```
remaining_guesses = 10 - num_guesses
puts remaining_guesses.to_s + " guesses left."
```

10 guesses left.

That works! Converting the number to a string first makes it clear to Ruby you're doing concatenation, not addition.

Ruby provides an easier way to handle this, though. Read on...

Ruby makes working with strings easy

Instead of calling to_s, we could save ourselves the effort of explicitly converting a number to a string by using string interpolation. As you saw in our code to greet the user, when you include # { } in a double-quoted string, code within the curly brackets is evaluated, converted to a string if necessary, and interpolated (substituted) into the longer string.

The automatic string conversion means we can get rid of the to s call.

```
remaining_guesses = 10 - num_guesses
puts "#{remaining_guesses} guesses left."
Ruby lets us do operations directly within the curly brackets, so we can also
get rid of the remaining_guesses variable.
    puts "#{10 - num_guesses} guesses left."
10 guesses left.
10 guesses
```

Now the player will know how many guesses they have left. We can check another requirement off our list!

```
# Get My Number Game
       # Written by: you!
       puts "Welcome to 'Get My Number!'"
       # Get the player's name, and greet them.
       print "What's your name? "
       input = gets
       name = input.chomp
       puts "Welcome, #{name}!"
       # Store a random number for the player to guess.
       puts "I've got a random number between 1 and 100."
       puts "Can you guess it?"
       target = rand(100) + 1
       # Track how many guesses the player has made.
      ) num_guesses = 0
Our new
codel
       puts "You've got #{10 - num guesses} guesses left."
```

Converting strings to numbers



Keep track of how many guesses the player has made. Before each guess, let them know how many guesses (out of 10) they have left.

Prompt the player to make a guess as to what the target number is.

Our next requirement is to prompt the player to guess the target number. So, we need to print a prompt, then record the user's input as their guess. The gets method, as you may recall, retrieves input from the user. (We already used it to get the player's name.) Unfortunately, we can't just use gets by itself to get a number from the user, because it returns a string. The problem will arise later, when we try to compare the player's guess with the target number using the > and < operators.

```
print "Make a guess: "
quess = gets
guess < target < Either of these will
guess > target < Fisher of these will
```

We need to convert the string returned from the gets method to a number so that we can compare the guess to our target number. No problem! Strings have a to i method to do the conversion for us.

This code will call to i on the string returned from gets. We don't even need to put the string in a variable first; we'll just use the dot operator to call the method directly on the return value.

quess = gets.to i

If we want to test our changes, we can print out the result of a comparison.

```
puts guess < target
```

true

Much better - we have a guess that we can compare to the target. That's another requirement done!

Common conversions

lf you call this method on an object	you get this kind of object back.
to_s	string
to_i	integer
to_f	floating-point number

```
# Store a random number for the player to guess.
          puts "I've got a random number between 1 and 100."
          puts "Can you guess it?"
          target = rand(100) + 1
          # Track how many guesses the player has made.
          num guesses = 0
Our new
          puts "You've got #{10 - num guesses} guesses left."
codel
          print "Make a guess: "
          quess = gets.to i
```

Conditionals

Two more requirements for our game down, four to go! Let's look at the next batch.

Prompt the player to make a guess as to what the target number is.
 If the player's guess is less than the target number, say "Oops. Your guess was LOW." If the player's guess is greater than the target number, say "Oops. Your guess was HIGH."
 If the player's guess is equal to the target number, tell them "Good job, [name]! You guessed my number in [number of guesses] guesses!"
 If the player runs out of turns without guessing correctly, say "Sorry. You didn't get my number. My number was [target]."

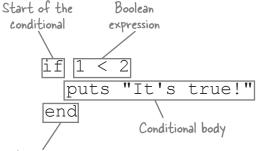
Now, we need to compare the player's guess with the target. *If* it's too high, we print a message saying so. *Otherwise, if* it's too low, we print a message to that effect, and so on... Looks like we need the ability to execute portions of our code only under certain *conditions*.

Like most languages, Ruby has **conditional** statements: statements that cause code to be executed only if a condition is met. An expression is evaluated, and if its result is true, the code in the conditional body is executed. If not, it's skipped.

Note that there's

no "e" in the

middle of "elsif





if score == 100
 puts "Perfect!"

else

end

elsif score >= 70

puts "You pass!"

puts "Summer school time!"

As with most other languages, Ruby supports multiple branches in the condition. These statements take the form if/elsif/else.

Conditionals rely on a **boolean**

expression (one with a true or false value) to decide whether the code they contain should be executed. Ruby has constants representing the two boolean values, true and false.

```
if true if false
  puts "I'll be printed!" puts "I won't!"
end end
```

Conditionals (cont.)

Ruby also has all the comparison operators you're used to.

if 1 == 1
 puts "I'll be printed!"
end

if 1 > 2
 puts "I won't!"
end

if 1 < 2
 puts "I'll be printed!"
end</pre>

It has the boolean negation operator, !, which lets you take a true value and make it false, or a false value and make it true. It also has the more-readable keyword not, which does basically the same thing.

If you need to ensure that two conditions are *both* true, you can use the && operator. If you need to ensure that *either* of two conditions are true, you can use the || operator.

00

if ! true
 puts "I won't be printed!"
end

if ! false puts "I will!" end

if true && true
 puts "I'll be printed!"
end

if true && false
 puts "I won't!"
end

if 1 >= 2
 puts "I won't!"
end

if 2 <= 2
 puts "I'll be printed!"
end
Said aloud as
if 2 != 2
 puts "I won't!"
end</pre>

if not true
 puts "I won't be printed!"
end

if not false puts "I will!" end

if false || true
 puts "I'll be printed!"
end

if false || false
 puts "I won't!"
end

I notice that you're indenting the code between the if and the end. Is that required?

Indented _____ puts "I'll be printed!" 2 spaces! end

Ruby doesn't treat indentation as significant to the meaning of the program, no. (Unlike some other languages, such as Python.)

But indenting code within if statements, loops, methods, classes, and the like is just good coding style. It helps make the structure of your code clear to your fellow developers (and even to yourself).

We need to compare the player's guess to the random target number. Let's use everything we've learned about conditionals to implement this batch of requirements.

```
# Get My Number Game
           # Written by: you!
           puts "Welcome to 'Get My Number!'"
                                                                        get_number.rb
We add this
           # Get the player's name, and greet them.
variable
           print "What's your name? "
to track
           input = gets
whether
           name = input.chomp
we should
           puts "Welcome, #{name}!"
print the
'you lost"
           # Store a random number for the player to guess.
message.
           puts "I've got a random number between 1 and 100."
We'll also
           puts "Can you guess it?"
use it later
           target = rand(100) + 1
to halt the
game on
           # Track how many guesses the player has made.
a correct
           num guesses = 0
quess.
           # Track whether the player has guessed correctly.
          🎙 quessed it = false
           puts "You've got #{10 - num guesses} guesses left."
           print "Make a guess: "
Here are
           quess = gets.to i
our "if"
statements
           # Compare the guess to the target.
           # Print the appropriate message.
           if guess < target
             puts "Oops. Your guess was LOW."
           elsif guess > target
             puts "Oops. Your guess was HIGH."
           elsif quess == target
We'll see a
             puts "Good job, #{name}!"
cleaner way
             puts "You guessed my number in #{num guesses} guesses!"
to write
             guessed it = true
this in a
           end
moment.
           # If player ran out of turns, tell them what the number was.
           if not guessed it
             puts "Sorry. You didn't get my number. (It was #{target}.)"
           end
```

The opposite of "if" is "unless"

This statement works, but it's a little awkward to read:

```
if not guessed_it
   puts "Sorry. You didn't get my number. (It was #{target}.)"
end
```

In most respects, Ruby's conditional statements are just like most other languages. Ruby has an additional keyword, though: unless.

Code within an if statement executes only if a condition is *true*, but code within an unless statement executes only if the condition is *false*.

unless	true	unless	false
puts	"I won't be printed!"	puts	"I will!"
end		end	

The unless keyword is an example of how Ruby works hard to make your code a little easier to read. You can use unless in situations where a negation operator would be awkward. So instead of this:

```
if ! light == "red"
   puts "Go!"
end
```

You can write this:

```
unless light == "red"
  puts "Go!"
end
```

We can use unless to clean up that last conditional.

```
unless guessed_it
   puts "Sorry. You didn't get my number. (It was #{target}.)"
end
```

Much more legible! And our conditional statements are working great!

You'll see something like this if you run get_number.rb now...

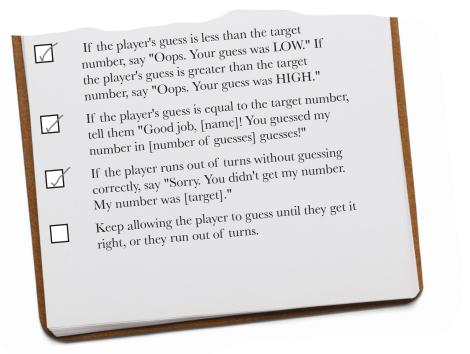
As it stands right now, though, the player only gets one guess - they're supposed to get 10. We'll fix that next...

Conventiona] * Wisdom 🔺 It's valid to use else and elsif together with unless in Ruby: unless light == "red" puts "Go!" else Confusing puts "Stop!" end But it's very hard to read. If you need an else clause, use if for the main clause instead! if light == "red" puts "Stop!" else Moved puts "Go!" up here. end

File Edit Window Help
<pre>\$ ruby get_number.rb</pre>
Welcome to 'Get My Number!'
What's your name? Jay
Welcome, Jay!
I've got a random number between 1 and 100.
Can you guess it?
You've got 10 guesses left.
Make a guess: 50
Oops. Your guess was HIGH.
Sorry. You didn't get my number. (It was 34.)

Loops

Great work so far! We have just one more requirement to go for our guessing game!



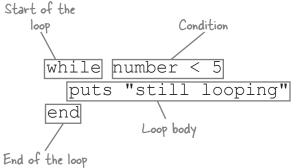
Currently, the player gets one guess. Since there's 100 possible target numbers, those don't seem like very fair odds. We need to keep asking them 10 times, or until they get the right answer, whichever comes first.

The code to prompt for a guess is already in place, we just need to run it *more than once*. We can use a **loop** to execute a segment of code repeatedly. You've probably encountered loops in other languages. When you need one or more statements to be executed over and over, you place them inside a loop.

A while loop consists of the word while, a boolean expression (just like in if or unless statements), the code you want to repeat, and the word end. The code within the loop body repeats *while* the condition is true.

Here's a simple example that uses a loop for counting.

number = 1	1
while number <= 5	2
	3
puts number	4
number += 1	5
end	



Just as unless is the counterpart to if, Ruby offers an until loop as a counterpart to while. An until loop repeats *until* the condition is true (that is, it loops while it's false).

Here's a similar example, using until.

```
number = 1
until number > 5
puts number
number += 1
end
```

Here's our conditional code again, updated to run within a while loop:

```
The loop will stop
                   # Track how many guesses the player has made.
after the player's
                   num quesses = 0
tenth quess, or
when they guess
                   # Track whether the player has guessed correctly.
correctly, whichever
                   quessed it = false
comes first.

ightarrowwhile num guesses < 10 && guessed it == false
   This code is
                     puts "You've got #{10 - num guesses} guesses left."
   exactly the
                     print "Make a guess: "
   same; we've just S
    nested it inside
                     guess = gets.to i
    the loop.
                  _∍ num_guesses += 1
We need to add -
I to the guess
                     # Compare the guess to the target.
count each loop,
                    # Print the appropriate message.
so we don't
                   if quess < target
loop forever.
                      puts "Oops. Your guess was LOW."
                    elsif quess > target
   No changes
                     puts "Oops. Your guess was HIGH."
   here, either.
                     elsif guess == target
                      puts "Good job, #{name}!"
                       puts "You guessed my number in #{num guesses} guesses!"
                       quessed it = true
                     end
 This marks the
 end of the -
                 为 end
 code that will
 loop.
                   unless quessed it
                     puts "Sorry. You didn't get my number. (It was #{target}.)"
                   end
```

There's one more readability improvement we can make. As with the if statement that we replaced with an unless, we can make this while loop read more clearly by replacing it with an until.

```
# Get My Number Game
Here's our
              # Written by: you!
complete
code listing.
              puts "Welcome to 'Get My Number!'"
                                                                            get_number.rb
              # Get the player's name, and greet them.
              print "What's your name? "
              input = gets
              name = input.chomp
              puts "Welcome, #{name}!"
              # Store a random number for the player to guess.
              puts "I've got a random number between 1 and 100."
              puts "Can you quess it?"
              target = rand(100) + 1
              # Track how many guesses the player has made.
              num guesses = 0
              # Track whether the player has guessed correctly.
              guessed it = false
              until num quesses == 10 || quessed it
               puts "You've got #{10 - num guesses} guesses left."
               print "Make a guess: "
                guess = gets.to i
                num guesses += 1
                # Compare the guess to the target.
                # Print the appropriate message.
                if guess < target
                  puts "Oops. Your guess was LOW."
                elsif guess > target
                  puts "Oops. Your guess was HIGH."
                elsif guess == target
                  puts "Good job, #{name}!"
                  puts "You guessed my number in #{num guesses} guesses!"
                  guessed it = true
                end
              end
              # If the player didn't guess in time, show the target number.
              unless guessed it
                puts "Sorry. You didn't get my number. (It was #{target}.)"
              end
```

Let's try running our game!

Our loop is in place - that's the last requirement! Let's open a command prompt, and try running the program!



Keep allowing the player to guess until they get it right, or they run out of turns.

File Edit Window Help Cheats]
<pre>\$ ruby get_number.rb</pre>	
Welcome to 'Get My Number!'	
What's your name? Gary	
Welcome, Gary!	
I've got a random number between 1 and 100.	
Can you guess it?	
You've got 10 guesses left.	
Make a guess: 50	
Oops. Your guess was LOW.	
You've got 9 guesses left.	
Make a guess: 75	
Oops. Your guess was HIGH.	\frown
You've got 8 guesses left.	
Make a guess: 62	
Oops. Your guess was HIGH.	Our players will love
You've got 7 guesses left.	/ this! You implemented /
Make a guess: 56	everything we needed, and
Oops. Your guess was HIGH.	you did it on time, too!
You've got 6 guesses left.	
Make a guess: 53	
Good job, Gary! You guessed my number in 5 guesses!	
\$	
Y	

Using variables, strings, method calls, conditionals, and loops, you've written a complete game in Ruby! Better yet, it took less than 30 lines of code! Pour yourself a cold drink - you've earned it!





Your Ruby Toolbox

You've got Chapter 1 under your belt and now you've added method calls, conditionals, and loops to your tool box.

Statements

Conditional statements execute the code they enclose if a condition is met.

Loops execute the code they enclose repeatedly. They exit when a condition is met.

BULLET POINTS

- Ruby is an interpreted language. You don't have to compile Ruby code before executing it.
- You don't need to declare variables before assigning to them. You also don't have to specify a type.
- Ruby treats everything from a # to the end of the line as a comment - and ignores it.
- Text within quotation marks is treated as a string - a series of characters.
- If you include #{} in a Ruby string, the expression in the brackets will be interpolated into the string.
- Method calls *may* need one or more arguments, separated by commas.
- Parenthesis are optional around method arguments. Leave them off if you're not passing any arguments.
- Use the inspect and p methods to view debug output for Ruby objects.
- You can include special characters within double-guoted strings by using escape sequences like \n and \t .
- You can use the interactive Ruby interpreter, or irb, to quickly test out the result of Ruby expressions.
- Call to s on almost any object to convert it to a string. Call to i on a string to convert it to an integer.
- unless is the opposite of if; its code won't execute unless a statement is false.
- until is the opposite of while; it executes repeatedly until a condition is true.

2 methods and classes

Getting Organized

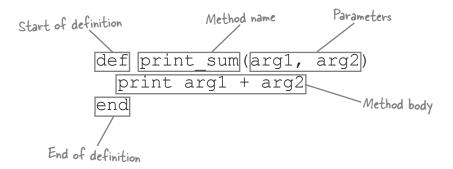


You've been missing out. You've been calling methods and creating objects like a pro. But the only methods you could call, and the only kinds of objects you could create, were the ones that Ruby defined for you. Now, it's your turn. You're going to learn to create your *own* methods. You'll also create your own **classes** - templates for new objects. *You'll decide* what objects based on your class will be like. You'll use **instance variables** to define what they *know*, and **instance methods** to define what they *do*. And most importantly, you'll discover how defining your own classes can make your code *easier to read and maintain*.

Pefining methods

Got-A-Motor, Inc. is working on their "virtual test-drive" app, which lets their customers try vehicles out on their computers without needing to visit a show room. For this first version, they need methods to let users step on the virtual gas, sound the virtual horn, and turn on the virtual headlights in low-beam or high-beam modes.

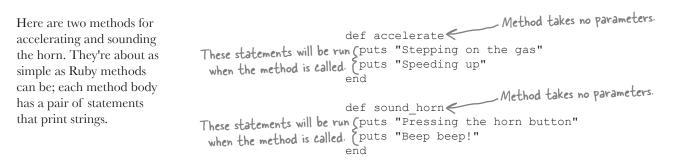
Method definitions look like this in Ruby:



If you want calls to your method to include arguments, you'll need to add **parameters** to the method definition. Parameters appear after the method name, within parenthesis. (You should leave off the parenthesis if there are no parameters.) Each argument on the method call gets stored in one of the parameters within the method.

The method body consists of one or more Ruby statements that are executed when the method is called.

Let's create our very own methods to represent the actions in the test-drive app.



The use_headlights method is only slightly more complex; it takes a single parameter, which is interpolated into one of the output strings.

That's all it takes! With these method definitions in place, we're ready to make calls to them.

Calling methods you've defined

You can call methods you've defined just like any other. Let's try out our new vehicle simulator methods.

Ruby lets you put calls to your methods anywhere - even within the same source file where you defined them. Since this is such a simple program at this point, we'll do that, just for convenience. We'll just stick the method calls right after the method declarations.

When we run the source file from the command line, we'll see the result of our method calls!

00

def accelerate puts "Stepping on the gas" puts "Speeding up" end def sound horn puts "Pressing the horn button" puts "Beep beep!" end def use headlights(brightness) puts "Turning on #{brightness} headlights" puts "Watch out for deer!" end Calls without sound_horn arguments. This is used as the accelerate -"brightness" argument. use headlights ("hi-beam") <vehicle methods.rb File Edit Window Help \$ ruby vehicle methods.rb Pressing the horn button Beep beep! Stepping on the gas Speeding up Turning on hi-beam headlights Watch out for deer! I notice you didn't use the dot operator to specify a receiver for those method calls, just like when we call the puts and print methods. That's right. Like puts and print, these methods are included in the top-level execution environment. Methods that are defined outside of any class (like these examples) are included in the top-level execution environment. Like we saw back in Chapter 1, you can call

them anywhere in your code, without using the dot operator to specify a receiver.

Method names

The method name can be one or more lower-case words, separated by underscores. (This is just like the convention for variable names.) Numbers are legal, but rarely used.

It's also legal for a method name to end in a question mark (?) or exclamation point (!). These endings have no special meaning to Ruby, but there are certain conventions around their use, which we'll cover in later chapters.

Lastly, it's legal for a method name to end in an equals sign (=). Methods ending in this character are used as attribute writers, which we'll be looking at in the upcoming section on classes. Ruby *does* treat this ending specially, so don't use it for a regular method, or you may find it acts strangely!

Parameters

If you need to pass data into your method, you can include one or more parameters after the method name, separated by commas. In your method body, parameters can be accessed just like any variable.

```
def print_area(length, width)
   puts length * width
end
```

Optional parameters

Got-A-Motor's developers are happy with our work on the virtual test drive system... mostly.





Method names should be in "snake case": one or more lower-case words, separated by underscores, just like variable names.

```
def bark
end
```

```
def wag_tail
end
```

As with method calls, you should leave parenthesis off the method definition if there are no parameters. Please <u>don't</u> do this, even though it's legal:

```
def no_args()
   puts "Bad Rubyist!"
end
```

But if there are parameters, you should always include parenthesis. (Back in Chapter 1, we showed some tasteful exceptions when making method calls, but there are no exceptions when <u>declaring</u> methods.) Leaving them off is legal, but again, <u>don't do it</u>:

```
def with_args first, second
   puts "No! Bad!"
end
```

use_headlights("<u>low-beam</u>")
stop_engine
buy_coffee
start_engine
use_headlights("<u>low-beam</u>")
accelerate
create_obstacle("deer")
use headlights("high-beam")

Optional parameters (cont.)

This scenario is pretty common - you use one particular argument 90% of the time, and you're tired of repeating it everywhere. You can't just take the parameter out, though, because 10% of the time you need a different value.

There's an easy solution, though; *make the parameter optional*. You can provide a default value in the method declaration.

Here's an example of a method that uses default values for some of its parameters:

```
def order_soda(flavor, size = "medium", quantity = 1)
    if quantity == 1
        plural = "soda"
    else
        plural = "sodas"
    end
    puts "#{quantity} #{size} #{flavor} #{plural}, coming right up!"
end
```

Now, if you want to override the default, just provide an argument with the value you want. And if you're happy with the default, you can skip the argument altogether.

```
order_soda ("orange")

order_soda ("lemon-lime", "small", 2)

Specify flavor, use default

for size and quantity.

Specify everything.

Specify flavor and size,

use default for quantity.
```

1 medium orange soda, coming right up! 2 small lemon-lime sodas, coming right up! 1 large grape soda, coming right up!

There is one caveat to be aware of with optional parameters: they need to appear *after* any other parameters you intend to use. If you make a required parameter following an optional parameter, you won't be able to leave the optional parameter off:

```
def order_soda (flavor, size = "medium", quantity)

... Don't place an optional parameter

end before a required one!

order_soda ("grape")

Error \longrightarrow wrong number of

arguments (1 for 2..3)
```

there are no Dumb Questions

Q: What's the difference between an argument and a parameter?

A: You define and use *parameters* within a method *definition*. You provide *arguments* with method *calls*.

Each argument you pass with the method call gets stored in a method parameter.

The two terms mostly serve to distinguish whether you're talking about a method *definition*, or a method *call*.

Optional parameters (cont.)

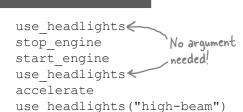
Let's earn some goodwill with the developers using our methods and make that use headlights parameter optional.

```
def use_headlights(brightness = "low-beam")
  puts "Turning on #{brightness} headlights"
  puts "Watch out for deer!"
end
```

Now, they won't have to specify the brightness, unless they want the high-beams.

use_headlights ("high-beam") Coverrides the default, "low-beam"

Turning on low-beam headlights Watch out for deer! Turning on high-beam headlights Watch out for deer!







We've finished up our methods for Got-A-Motor's virtual test drive app. Let's try loading them up in irb, and take them for a spin.

Exercise Step One:	Save our method definitions to a file, named	def accelerate puts "Stepping on the gas" puts "Speeding up" end vehicle_methods.rb
Step Two:	"vehicle_methods.rb". Open a system command prompt, and navigate into	def sound_horn puts "Pressing the horn button" puts "Beep beep!" end
	the directory where you saved your file.	<pre>def use_headlights(brightness = "low-beam") puts "Turning on #{brightness} headlights" puts "Watch out for deer!" end</pre>



Step Three:

Since we're loading code from a file into irb, we want to be able to load Ruby files from the current directory. So we're going to invoke irb a little differently this time.

At the command prompt, type this and press Enter:

irb <u>-I</u>. A flag that means "search the current directory for files to load".

The -I is a *command line flag*, a string that you add on to a command to change how it operates. In this case, -I alters the set of directories that Ruby searches for files to load. And the dot (.) represents the current directory.

Step Four: Now, irb should be loaded, and we should be able to load the file with our methods. Type this line:

```
require "vehicle methods"
```

Ruby knows to search in .rb files by default, so you can leave the extension off. If you see the result true, it means your file was loaded successfully.

```
Now, you can type in a
                        Here's a
                                    File Edit Window Help
                        sample session:
call to any of our methods,
                                    $ irb -I .
and they'll be run!
                                    irb(main):001:0> require "vehicle methods"
                                     => true
                                     irb(main):002:0> sound horn
                                    Pressing the horn button
                                    Beep beep!
                                     => nil
                                    irb(main):003:0> use headlights
                                    Turning on low-beam headlights
                                    Watch out for deer!
                                     \Rightarrow nil
                                    irb(main):004:0> use headlights("high-beam")
                                    Turning on high-beam headlights
                                    Watch out for deer!
                                     => nil
                                    irb(main):005:0> exit
                                    $
```

Return value

Got-A-Motor wants the test-drive app to highlight how fuel-efficient its cars are. They want to be able to display the mileage a car got on its most recent trip, as well as lifetime average mileage.

In the first scenario, you're dividing the mileage from the car's trip odometer by the number of gallons from your last fillup, and in the second you're dividing the main odometer's value by the car's lifetime fuel use. But in both cases, you're taking a number of miles, and dividing it by a number of gallons of fuel. So, do you still have to write two methods?

Nope! Like in most languages, Ruby methods have a **return value**, a value that they can send back to the code that called them. A Ruby method can return a value to its caller using the return keyword.

You can write a single mileage method, and use its return value in your output.

```
def mileage(miles_driven, gas_used)
  return miles_driven / gas_used
end
```

Then, you can use the same method to calculate both types of mileage.

```
trip_mileage = mileage(400, 12)
puts "You got #{trip_mileage} MPG on this trip."
lifetime_mileage = mileage(11432, 366)
puts "This car averages #{lifetime_mileage} MPG."
```

You got 33 MPG on this trip. This car averages 31 MPG.

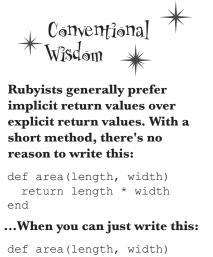
Implicit return values

You don't actually need the return keyword in the above method. The value of the last expression evaluated within a method automatically becomes that method's return value. So, our mileage method could be rewritten without an explicit return:

```
def mileage(miles_driven, gas_used)
  miles_driven / gas_used
end
```

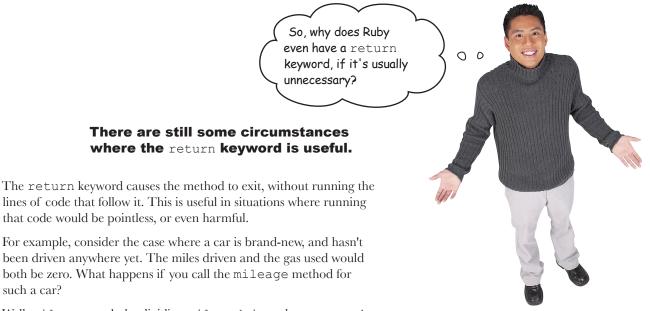
It will still work in exactly the same way.

puts mileage(400, 12) 33



def area(length, width)
 length * width
end

Returning from a method early



Well, mileage works by dividing miles driven by gas used ... And as you may have learned in your other programming language, dividing anything by zero is an error!

puts mileage(0, 0)

such a car²

in `/': divided by 0 (ZeroDivisionError)

We can fix this by testing whether gas used is zero, and if so, returning from the method early.

```
def mileage(miles driven, gas used)
 if gas_used == 0 (If no gas has been used...
return 0.0 (Return zero.
 end
 end
```

Error ----

If we try the same code again, we'll see that it returns 0.0, without attempting the division operation. Problem solved!

puts mileage(0, 0)

Methods are a great way to reduce duplication, and keep your code organized. But sometimes, methods by themselves aren't enough. Let's leave our friends at Got-A-Motor for now, to look at a somewhat fuzzier problem...

0.0

Some messy methods

Fuzzy Friends Animal Rescue is in the middle of a fundraising drive, and are doing an interactive storybook application to raise awareness. They've approached your company for help. They need many different types of animals, each of which has its own sounds and actions.

They've created some methods that simulate movement and animal noises. Their methods are called by specifying the animal type as the first argument, followed by any additional arguments that are needed.

Here's what they have so far:

```
def talk(animal type, name)
  if animal type == "bird"
    puts "#{name} says Chirp! Chirp!"
  elsif animal type == "dog"
    puts "#{name} says Bark!"
                                    The animal type
  elsif animal type == "cat"
                                   parameter is used
    puts "#{name} says Meow!"
                                    to select which
  end
                                    string is printed.
end
def move (animal type, name, destination)
  if animal type == "bird"
    puts "#{name} flies to the #{destination}."
  elsif animal type == "dog"
    puts "#{name} runs to the #{destination}."
  elsif animal type == "cat"
    puts "#{name} runs to the #{destination}."
                              This method is the same for
  end
end
                               all animal types, so there's
def report_age (name, age) - no animal type parameter.
  puts "#{name} is #{age} years old."
end
```

And here are some typical calls to those methods:

move("bird", "Whistler", "tree")
talk("dog", "Sadie")
talk("bird", "Whistler")
move("cat", "Smudge", "house")
report_age("Smudge", 6)

Whistler flies to the tree. Sadie says Bark! Whistler says Chirp! Chirp! Smudge runs to the house. Smudge is 6 years old.

Fuzzy Friends just needs you to add 10 additional animal types and 30 more actions, and version 1.0 will be done!

Too many arguments

That's looking pretty messy with just three animal types and two actions. Those "if" and "elsif" statements are long already, and look at 0 all those method arguments! Isn't there a better way to organize this code? Part of the problem with the virtual storybook methods is that We need the we're having to pass around too much data. Look at these calls to destination argument. the move method, for example: move("bird", "Whistler" "tree" move ("cat", "Smudge", ("house") ... But do we really have to pass these each time? The destination argument belongs there, sure. It doesn't make sense to move without a destination. But do we really have to keep track of values for the animal type and name arguments, so that we can include them each time? It's also becoming hard to tell which argument is which!

Too many "if" statements

The problem isn't just with the method arguments, either – things are messy *inside* the methods. Consider what the talk method would look like if we added ten more animal types, for example...

Each time you want to change the sound an animal makes (and you *will* be asked to change the sounds, you can count on it), you'll have to search through all those elsif clauses to find the right animal type... What happens when the code for talk becomes more complex, adding things like animations and sound file playback? What happens when *all* of the action methods are like that?

What we need is a better way to represent which animal type we're working with. We need a better way to break all that code up by animal type, so that we can maintain it more easily. And we need a better way to store the attributes for each individual animal, like their name and their age, so we don't have to pass so many arguments around.

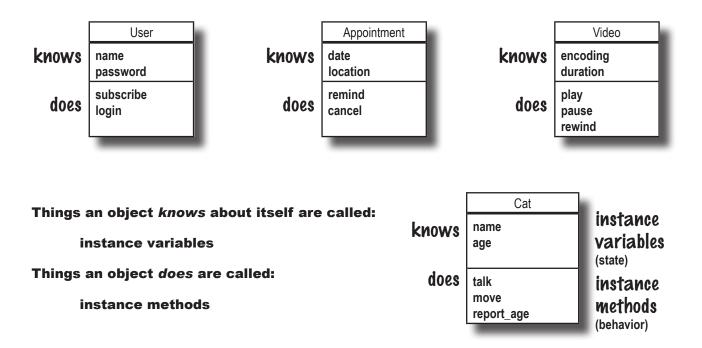
We need to keep the animals' data, and the code that operates on that data, in one place. We need: *classes* and *objects*.

```
def talk(animal type, name)
  if animal type == "bird"
    puts "#{name} says Chirp! Chirp!"
  elsif animal type == "dog"
    puts "#{name} says Bark!"
  elsif animal type == "cat"
    puts "#{name} says Meow!"
  elsif animal type == "lion"
    puts "#{name} says Roar!"
  elsif animal type == "cow"
    puts "#{name} says Moo."
  elsif animal type == "bob"
    puts "#{name} says Hello."
  elsif animal type == "duck"
    puts "#{name} says Quack."
  ... We don't even have room
  end
            to print all this ...
end
```

Designing a class

The benefit of using objects is that they keep a set of data, and the methods that operate on that data, in one place. We want those benefits in the Fuzzy Friends app.

To start creating your own objects, though, you're going to need classes. A **class** is a blueprint for making objects. When you use a class to make an object, the class describes what that object *knows* about itself, as well as what that object *does*.



An **instance** of a class is an object that was made using that class. You only have to write *one* class, but you can make *many* instances of that class.

Think of "instance" as another way of saying "object".

Instance variables are variables that belong to one object. They comprise everything the object **knows** about itself. They represent the object's state (its data), and they can have different values for each instance of the class.

Instance methods are methods that you can call directly on that object. They comprise what the object **does**. They have access to the object's instance variables, and can use them to change their behavior based on the values in those variables.

What's the difference between a class and an object?

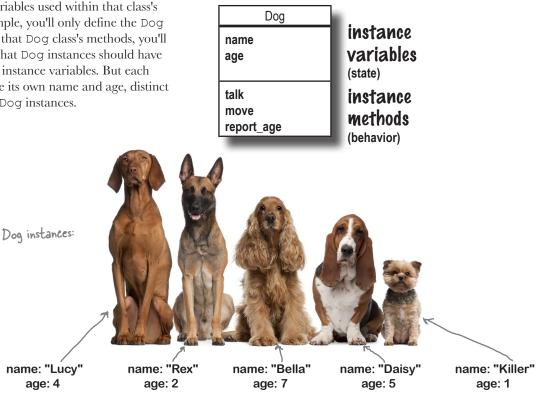
A class is a blueprint for an object. The class tells Ruby how to make an object of that particular type. Objects have instance variables and instance methods, but those variables and methods are designed as part of the class.

If classes are cookie cutters, objects are the cookies they make.



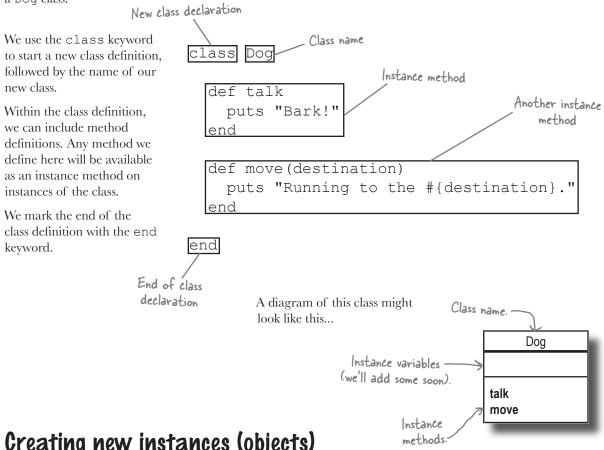
Each instance of a class can have its own values for the instance variables used within that class's methods. For example, you'll only define the Dog class once. Within that Dog class's methods, you'll only specify once that Dog instances should have "name" and "age" instance variables. But each Dog *object* will have its own name and age, distinct from all the other Dog instances.

Dog class:



Your first class

Here's an example of a class we could use in our interactive storybook: a Dog class.



Creating new instances (objects)

If we call the new method on a class, it will return a new instance of that class. We can then assign that instance to a variable, or whatever else we need to do with it.

> fido = Dog.new rex = Dog.new

Once we have one or more instances of the class, we can call their instance methods. We do it in the same way we've called all other methods on objects so far: we use the dot operator to specify which instance is the method's reciever.

> fido.talk rex.move("food bowl")

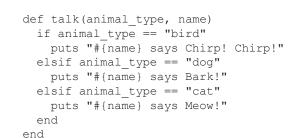
Bark! Running to the food bowl.

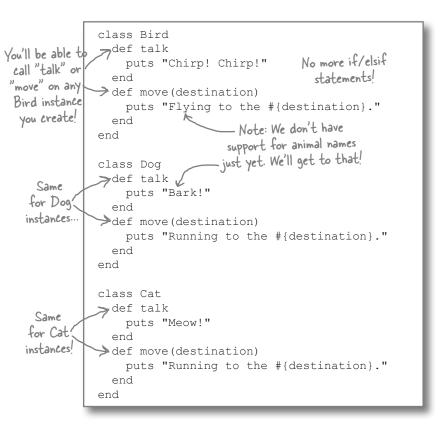
Breaking our giant methods up into classes

The animal rescue's solution uses strings to track what type of animal they're dealing with. Also, all knowledge of the different ways that different animals should respond is embedded in giant if/else statements. Their approach is unwieldy, at best.

The object-oriented approach

Now that you know how to create classes, we can take an *object-oriented* approach to the problem. We can create a *class* to represent each *type* of animal. Then, instead of one *big* method that contains behavior for *all* the animal types, we can put *little* methods in *each* class, methods that define behavior specific to that type of animal.





* Convențional * Wisdom *

Ruby class names must begin with a capital letter. Letters after the first should be lower case.

class Appointment

end

If there's more than one word in the name, the first letter of each word should also be capitalized.

```
class AddressBook
...
end
class PhoneNumber
...
end
```

Remember how the convention for variable names (with underscores separating words) is called "snake case"? The style for class names is called "camel case", because the capital letters look like the humps on a camel.

Creating instances of our new animal classes

With these classes defined, we can create new instances of them (new objects based on the classes), and call methods on them.

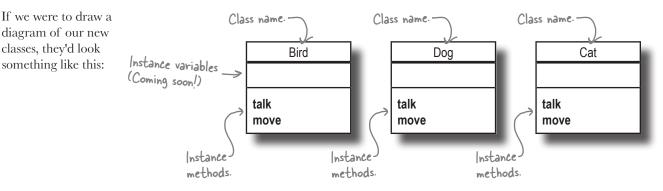
Just as with methods, Ruby lets us create instances of classes right in the same file where we declared them. You probably won't want to organize your code this way in larger applications, but since this is such a simple app right now, we can go ahead and create some new instances right below the class declarations.

```
class Bird
  def talk
   puts "Chirp! Chirp!"
  end
                                 animals.rb
  def move (destination)
   puts "Flying to the #{destination}."
  end
end
class Dog
  def talk
   puts "Bark!"
  end
  def move (destination)
  puts "Running to the #{destination}."
  end
end
class Cat
  def talk
   puts "Meow!"
  end
  def move (destination)
   puts "Running to the #{destination}."
  end
end
bird = Bird.new)
                Create new instances
dog = Dog.new
cat = Cat.new ) of our classes.
bird.move("tree"))
dog.talk
                    Call some methods on
bird.talk
                       the instances
cat.move("house")
```

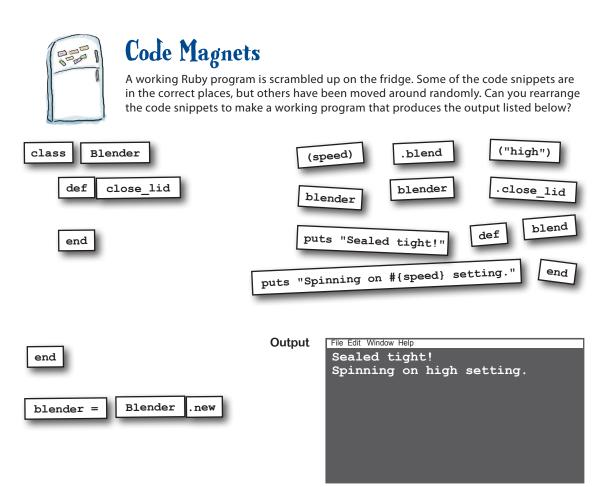
If we save all this to a file named animals.rb, then run ruby animals.rb at a command prompt, we'll see the output of our instance methods!

File Edit Window Help
\$ ruby animals.rb
Flying to the tree. Bark!
Chirp! Chirp!
Running to the house. \$

Updating our class diagram with instance methods



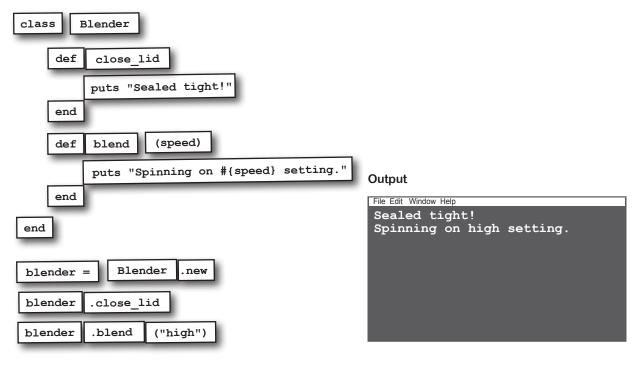
At this point, instances of our classes have two instance methods (things they can *do*): talk and move. They don't have any instance variables (things they *know*) yet, however. We'll be looking at that next.





Code Magnets Solution

A working Ruby program is scrambled up on the fridge. Some of the code snippets are in the correct places, but others have been moved around randomly. Can you rearrange the code snippets to make a working program that produces the output listed below?



bumb Questions

Q: Can I call these new move and talk methods by themselves (without an object)?

A: Not from outside the class, no. Remember, the purpose of specifying a receiver is to tell Ruby which object a method is being called on. The move and talk methods are *instance methods*; it doesn't make sense to call them without stating which instance of the class you're calling them on. If you try, you'll get an error, like this:

```
move("food bowl")
undefined method `move' for
main:Object (NoMethodError)
```

You say that we have to call the $n \in w$ method on a class to create an object. You also said back in chapter 1 that numbers and strings are objects. Why don't we have to call $n \in w$ to get a new number or string?

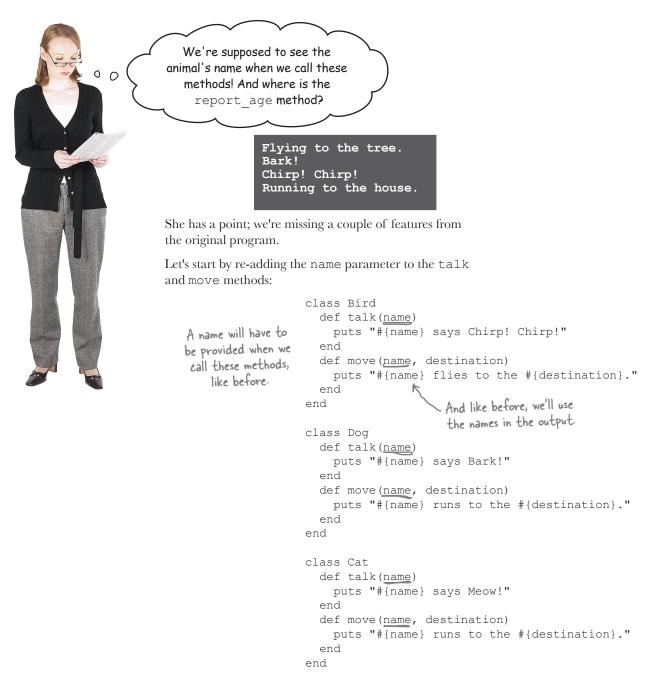
A: Creating new numbers and strings is something developers need to do so frequently that special shorthand notation is built right into the language: string and number *literals*.

```
new_string = "Hello!"
new float = 4.2
```

Doing the same for other classes would require modifying the Ruby language itself, so most of them just rely on new to create new instances. (There are exceptions; we'll get to those in later chapters.)

Our objects don't "know" their names or ages!

The animal rescue's lead developer points out a couple details we forgot to address with our class-based solution:

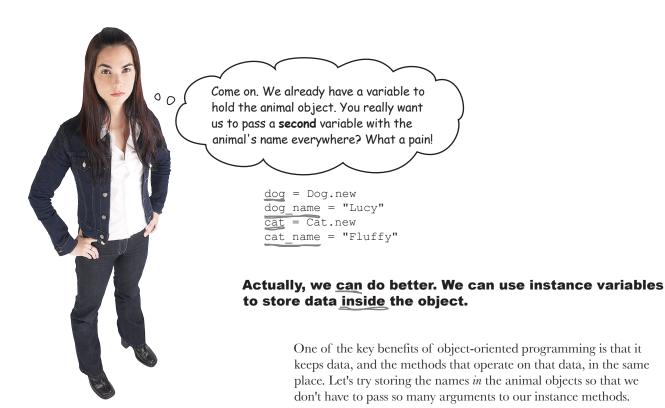


Too many arguments (again)

Now that we've re-added the name parameter to the talk and move methods, we can once again pass in the animal's name to be printed.

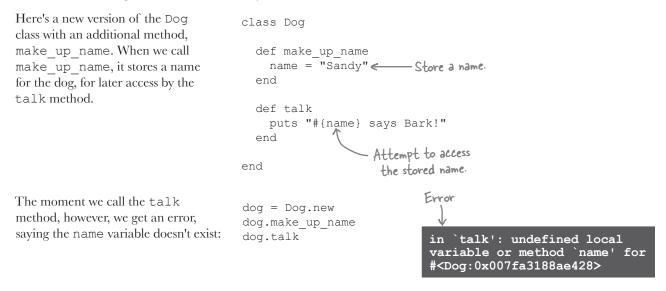
dog = Dog.new dog name = "Lucy" dog.talk(dog name) dog.move(dog name, "fence") cat = Cat.new cat name = "Fluffy" cat.talk(cat name) cat.move(cat name, "litter box")

Lucy says Bark! Lucy runs to the fence. Fluffy says Meow! Fluffy runs to the litter box.



Local variables live until the method ends

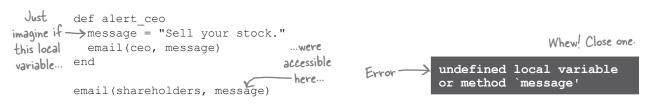
So far, we've been working with **local variables** - variables that are *local* to the current scope (usually the current method). When the current scope ends, local variables cease to exist, so they *won't* work for storing our animals' names, as you'll see below.



What happened? We *did* define a name variable, back in the make up name method!

The problem, though, is that we used a <i>local</i> variable. Local	
variables live only as long as the method in which they were	class Dog
created. In this case, The name variable ceases to exist as soon as make_up_name ends.	def make_up_name name = "Sandy" end \leftarrow name" drops out of scope as soon as the method ends
	def talk
	puts "#{name} says Bark!"
	end (
	This variable no
	end longer exists here!

Trust us, the short life of local variables is a *good* thing. If *any* variable was accessible *anywhere* in your program, you'd be accidentally referencing the wrong variables *all the time*! Like most languages, Ruby limits the scope of variables in order to prevent this sort of mistake.



Instance variables live as long as the instance does

Any local variable we create disappears as soon as its scope ends. If that's true, though, how can we store a Dog's name together with the object? We're going to need a new kind of variable.

An object can store data in **instance variables** - variables that are tied to a particular object instance. Data written to an object's instance variables stays with that object, getting removed from memory only when the object is removed.

An instance variable looks just like a regular variable, and follows all the same naming conventions. The only difference in syntax is that its name begins with an "at" symbol (@).

my variable @my variable

Local variable

Instance variable

Here's that Dog class again. It's identical to the previous one, except that we added two little "@" symbols to convert the *two* local variables to *one* instance variable.

class Dog def make_up_name Store a value in an ______ @name = "Sandy" instance variable def talk puts "#{@name} says Bark!" end _______ end ______ end ______ Access the instance variable.

Now, we can make the exact same call to talk that we did before, and the code will work! The @name instance variable that we create in the make up name method is still accessible in the talk method.

dog = Dog.new dog.make up name dog.talk Sandy says Bark!

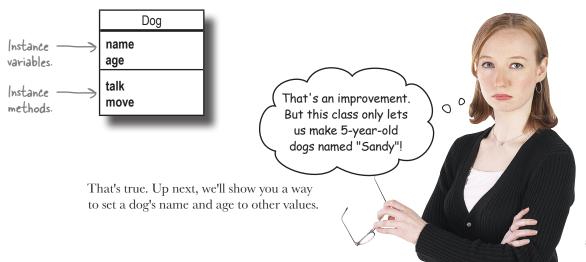
Instance variables live as long as the instance does (cont.)

With instance variables at our disposal, it's easy to add the move and report_age methods back in, as well...

```
class Dog
        def make up name
          @name = "Sandy"
        end
        def talk
          puts "#{@name} says Bark!"
        end
        def move(destination)
          puts "#{@name} runs to the #{destination}."
        end
        def make up age
Our new
          @age = 5
code
        end
        def report age
          puts "#{@name} is #{@age} years old."
        end
      end
      dog = Dog.new
      dog.make up name
      dog.move("yard")
      dog.make up age
      dog.report age
```

Sandy runs to the yard. <u>Sandy</u> is 5 years old.

And now that we have instance variables, we can finally fill in that hole in the class diagram for Dog!



Encapsulation

class Dog Thanks to instance variables, we now have a way to store names and ages for our animals. But our make up name and def make up name make up age methods only allow us to use hard-coded values @name = "Sandv" (we can't change them when the program's running). We need a end way for our program to set any values we want. def make up age Qage = 5end . . . end Error Code like this *won't* work, though: fido = Dog.new

fido.@age = 3

syntax error, unexpected tIVAR

Ruby never allows us to access instance variables directly from outside our class. This isn't due to some authoritarian agenda; it's to keep other programs and classes from modifying your instance variables willy-nilly.

Let's suppose that you *could* update instance variables directly. What's to prevent other portions of the program from setting the variables to invalid values?



Who is how old? This object's data is clearly invalid, and the user can see it in the program output!

Blank names and negative ages are just the start. Imagine someone accidentally replacing the value in an Appointment object's @date instance variable with a phone number. Or setting the @sales_tax on all their Invoice objects to zero. All kinds of things could go wrong!

To help avoid exposing an object's data to malicious (or clumsy) users, most objectoriented languages encourage the concept of **encapsulation**: of preventing other parts of the program from directly accessing or changing an object's instance variables.

Attribute accessor methods

To encourage encapsulation and protect your instances from invalid data, Ruby doesn't allow you to access or change instance variables from outside the class. Instead, you can create **accessor methods**, which will write values to the instance variables and read them back out again for you. Once you're accessing your data through accessor methods, it's easy to extend those methods to *validate* your data—to reject any bad values that get passed in.

Ruby has two kinds of accessor methods: *attribute writers* and *attribute readers*. (An "attribute" is another name for a piece of data regarding an object.) Attribute *writer* methods *set* an instance variable, and attribute *reader* methods *get* the value of an instance variable back.

Here's a simple class with writer and reader methods for an attribute named my_attribute:

Attribute class MyClass writer method (def my attribute=(new value)≪ @my attribute = new value end Accessor 5 def my_attribute methods. → Attribute @my attribute reader lend method. end If we create a new instance of the above class... my instance = MyClass.new

we can set the attribute like this	<pre>my_instance.my_attribute = "a value"</pre>
and read the attribute like this.	puts my_instance.my_attribute

Accessor methods are just ordinary instance methods; we only refer to them as "accessor methods" because their primary purpose is to access an instance variable.

Look at the attribute reader method, for example; it's a perfectly ordinary method that simply returns the current value of <code>@my_attribute</code>.

Attribute accessor methods (cont.)

Like attribute *reader* methods, an attribute *writer* method is a perfectly ordinary instance method. We just call it an "attribute writer" method because the primary thing it does is to update an instance variable.

It may be a perfectly ordinary method, but *calls* to it are treated somewhat specially.

Remember that earlier in the chapter, we said that Ruby method names could end in "="? Ruby allows that equals-sign ending so that it can be used in the names of attribute writer methods.

When Ruby sees something like this in your code:

my_instance.my_attribute = "a value"

...it gets translated into a call to the my_attribute= instance method. The value to the right of the "=" is passed as an argument to the method:

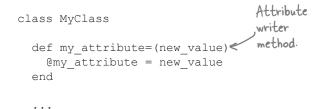
```
A method

call! The method

my instance.my_attribute=("a value")
```

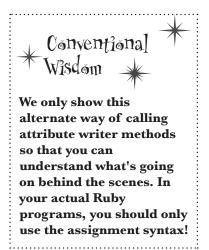
The above code is valid Ruby, and you can try it yourself, if you like:

```
class MyClass
  def my attribute=(new value)
    @my attribute = new value
  end
  def my attribute
    @my_attribute
  end
                              A call to "my_attribute=",
- disguised as assignment.
end
my instance = MyClass.new
my_instance.my_attribute = "assigned via method call"
                                               A call to
puts my instance.my attribute
my instance.my_attribute=("same here") / "my_attribute=
                                               that actually
puts my instance.my attribute
                                                looks like one!
```



end





assigned via method call same here

Using accessor methods

Now we're ready to use what we've learned in the Fuzzy Friends application. As a first step, let's update the Dog class with methods that will let us read and write @name and @age instance variables. We'll also use @name and @age in the report_age method. We'll look at adding data validation later.

```
class Dog
                           Write a new value
    @name = new_value to @name
  def name=(new value)
  end
  def name
    Cname Read the value
from Oname
  end
  def age=(new value)
    Cage = new_value Write a new value
to Cage
  end
  def age
    Cage Cage Read the value
                    from @age
  end
  def report age
    puts "#{@name} is #{@age} years old."
  end
end
```

With accessor methods in place, we can (indirectly) set and use the @name and @age instance variables from outside the Dog class!

```
fido = Dog.new Set @name for Fido.

fido.name = "Fido" Set @age for Fido.

rex = Dog.new

rex.name = "Rex" Set @name for Rex.

rex.age = 3 Set @age for Rex.

fido.report_age

rex.report_age
```

Fido is 2 years old. Rex is 3 years old.

Writing a reader and writer method by hand for each attribute can get tedious, though. Next, we'll look at an easier way...



The name of an attribute reader method should usually match the name of the instance variable it reads from (without the @ symbol, of course).

```
def tail_length
  @tail_length
end
```

The same is true for attribute writer methods, but you should add an = symbol on to the end of the name.

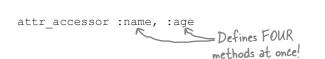
```
def tail_length=(value)
  @tail_length = value
end
```

Attribute writers and readers

Creating this pair of accessor methods for an attribute is so common that Ruby offers us shortcuts - methods named: attr_writer, attr_reader, and attr_accessor. Calling these three methods within your class definition will automatically define new accessor methods for you:

Write this within your class definition	and Ruby will automatically define these methods:	
attr_writer :name	def name=(new_value) @name = new_value end	Just like our old definition!
attr_reader :name	def name @name end	Just like — our old definition!
attr_accessor :name	<pre>def name=(new_value) @name = new_value end def name @name end</pre>	Defines two methods at once!

All three of these methods can take multiple arguments, specifying multiple attributes that you want to define accessors for.



Symbols

In case you're wondering, those :name and :age things are *symbols*. A Ruby **symbol** is a series of characters, like a string. Unlike a string, its value can't be changed later. That makes them perfect for use inside Ruby programs, to refer to anything whose name doesn't (usually) change, like a method. For example, if you call the method named methods on an object in irb,

you'll see that it returns a list of symbols.

A symbol reference in Ruby code always begins with a colon character (:). A symbol should be in all lower-case, with words separated by underscores, just like a variable name.

Attribute writers and readers in action

The Dog class currently devotes 12 lines of code to accessor methods. With the attr_accessor method, we can shrink that down to 1 line!

It will let us reduce our Dog class's size ...

```
from this...
                                                                 ...to this!
class Dog
                                             class Dog
                                 Equivalent!
 def name=(new value)-
                                               attr accessor :name, :age
    @name = new value
                                               def report age
 end
                                                 puts "#{@name} is #{@age} years old."
 def name
                                               end
                              Equivalent
    0name
 end
                                               def talk
                                                 puts "#{@name} says Bark!"
 def age=(new value)
                                               end
    @age = new value
                                               def move(destination)
 end
                                                 puts "#{@name} runs to the #{destination}."
 def age
                                               end
    @age
                                             end
 end
 def report age
   puts "#{@name} is #{@age} years old."
 end
 def talk
   puts "#{@name} says Bark!"
 end
 def move(destination)
    puts "#{@name} runs to the #{destination}."
 end
end
```

...how's that for efficiency? It's a lot easier to read, too!

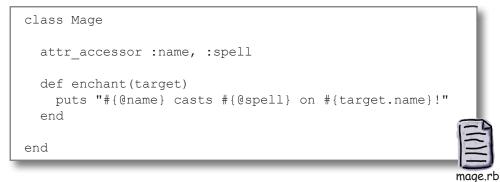
Let's not forget why we're writing accessor methods in the first place, though. We need to *protect* our instance variables from invalid data. Right now, these methods don't do that... We'll see how to fix this in a few pages!



We haven't really gotten to play around with classes and objects much yet. Let's try another irb session. We'll load up a simple class so we can create some instances of it interactively.

Step One:

Save this class definition to a file, named "mage.rb".



Step Two:

From a system command prompt, navigate into the directory where you saved your file.

Step Three:

We want to be able to load Ruby files from the current directory, so as in the previous exercise, type the following to launch irb:

irb -I .

Step Four:

As before, we need to load the file with our saved Ruby code. Type this line:

```
require "mage"
```



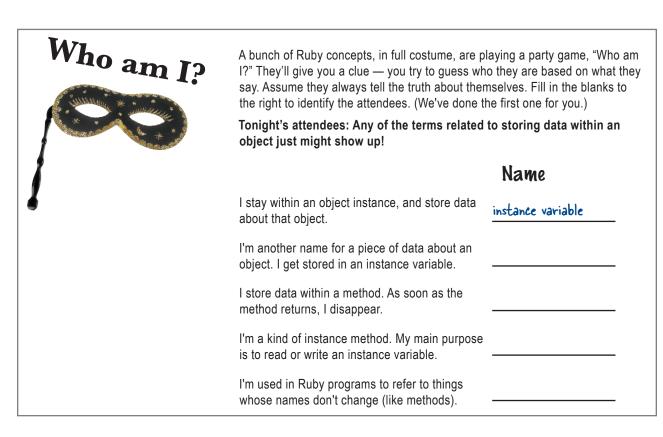
Here's a sample session:

File Edit Minder

With our Mage class's code loaded, you can try creating as many instances as you like, set their attributes, and have them cast spells at each other! Try the following for starters:

merlin = Mage.new
merlin.name = "Merlin"
morgana = Mage.new
morgana.name = "Morgana"
morgana.spell = "Shrink"
morgana.enchant(merlin)

\$ irb -I .
<pre>irb(main):001:0> require 'mage' => true</pre>
irb(main):002:0> merlin = Mage.new => # <mage:0x007fd432082308></mage:0x007fd432082308>
<pre>irb(main):003:0> merlin.name = "Merlin" => "Merlin"</pre>
irb(main):004:0> morgana = Mage.new => # <mage:0x007fd43206b310></mage:0x007fd43206b310>
<pre>irb(main):005:0> morgana.name = "Morgana" => "Morgana"</pre>
<pre>irb(main):006:0> morgana.spell = "Shrink" => "Shrink"</pre>
<pre>irb(main):007:0> morgana.enchant(merlin)</pre>
Morgana casts Shrink on Merlin! => nil
irb(main):008:0>



Who am I? Solution



I stay within an object instance, and store data about that object.

I'm another name for a piece of data about an object. I get stored in an instance variable.

I store data within a method. As soon as the method returns, I disappear.

I'm a kind of instance method. My main purpose is to read or write an instance variable.

I'm used in Ruby programs to refer to things whose names don't change (like methods).

Name

instance variable

attribute

local variable

accessor method

symbol

there lare no Dumb Questions

Q: What's the difference between an accessor method and an instance method?

A: "Accessor method" is just a way of describing one particular *kind* of instance method, one whose primary purpose is to get or set the value of an instance variable. In all other respects, accessor methods are ordinary instance methods.

Q: I set up an instance variable outside an instance method, but it's not there when I try to access it. Why?

```
class Widget
@size = 'large'
def show_size
   puts "Size: #{@size}"
   end
end
Empty!
```

widget = Widget.new
widget.show_size

Size:

A: When you use instance variables outside of an instance method, you're actually creating an instance variable on the class object. (That's right, even classes are themselves objects in Ruby.)

While there are potential uses for this, they're beyond the scope of this book. For now, this is almost certainly not what you want. Instead, set up the instance variable within an instance method:

```
class Widget
  def set_size
    @size = 'large'
  end
   ...
end
```

Pool Puzzle

T

Your **job** is to take code snippets from the pool and place them into the blank lines in the code. Don't use the same snippet more than once, and you won't need to use all the snippets. Your goal is to make code that will run and produce the output shown.

	class Robot	robot = Robot.new
	def @head	robot.assemble
	end	robot.arms = "MagGrip Claws" robot.eyes = "X-Ray Scopes"
	def (value) @arms = value	robot.feet = "MagGrip Boots"
	end	puts robot.head puts robot.legs
	:legs, :body	puts robot.body puts robot.feet
	attr_writer	robot.diagnostic
	:feet	Output
	<pre>def assemble @legs = "RubyTek Walkers" @body = "BurlyBot Frame" = "SuperAI 9000" end def diagnostic puts puts @eyes end end</pre>	File Edit Window Help SuperAI 9000 RubyTek Walkers BurlyBot Frame MagGrip Boots MagGrip Claws X-Ray Scopes
Note: each thing from the pool can only be used once!	@arms arms= attr_r	eader
attr_ac	attr_writer cessor :eyes	@head @feet :head head



Pool Puzzle Solution

Your **job** is to take code snippets from the pool and place them into the blank lines in the code. **Don't** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make code that will run and produce the output shown.

class Robot

def <u>head</u> @head end

```
def arms=(value)
  @arms = value
end
```

attr reader :legs, :body

```
attr writer :eyes
```

puts @arms puts @eyes

attr_accessor :feet

```
def assemble
  @legs = "RubyTek Walkers"
  @body = "BurlyBot Frame"
  @head = "SuperAI 9000"
end
def diagnostic
```

```
robot.assemble
robot.arms = "MagGrip Claws"
robot.eyes = "X-Ray Scopes"
robot.feet = "MagGrip Boots"
puts robot.head
puts robot.legs
puts robot.legs
puts robot.body
puts robot.feet
robot.diagnostic
```

robot = Robot.new

Output

File Edit Window Help Lasers SuperAI 9000 RubyTek Walkers BurlyBot Frame MagGrip Boots MagGrip Claws X-Ray Scopes

end

end

Ensuring data is valid with accessors

Remember our scenario from a nightmare world where Ruby let programs access instance variables directly, and someone gave your Dog instances *blank* names and *negative* ages? Bad news: now that you've added attribute writer methods to your Dog class, they actually *can*!

```
joey = Dog.new
joey.name = ""
joey.age = -1
joey.report_age
```

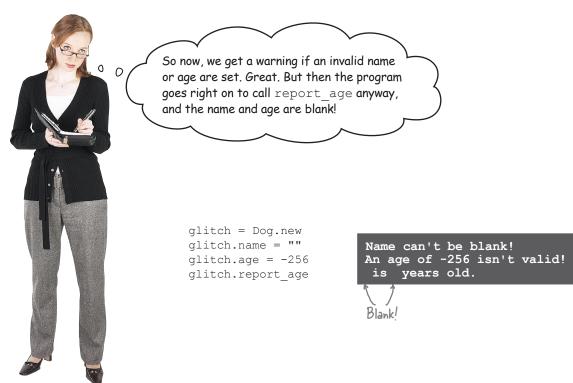
Don't panic! Those same writer methods are going to help us prevent this from happening in the future. We're going to add some simple data *validation* to the methods, which will give an error any time an invalid value is passed in.

Since name= and age= are just ordinary Ruby methods, adding the validation is really easy; we'll use ordinary if statements to look for an empty string (for name=) or a negative number (for age=). If we encounter an invalid value, we'll print an error message. Only if the value is valid will we actually set the @name or @age instance variables.

```
We only define the
                         reader methods
                         automatically, since
                         we're defining writer
                         methods ourselves.
class Dog
  attr reader :name, :age
                                If the name is blank,
                               print an error message
  def name=(value)
    if value == ""
       puts "Name can't be blank!"
    else
       @name = value Set the instance variable
                              only if the name is valid.
    end
  end
                            |f the age is negative,
— print an error message.
  def age=(value)
    if value < 0 ←
       puts "An age of #{value} isn't valid!"
    else
       lage = value Set the instance variable
    end
                            only if the age is valid.
  end
  def report age
    puts "#{@name} is #{@age} years old."
  end
```

is -1 years old.

Errors - the "emergency stop" button



Instead of just *printing* a message, we need to deal with invalid parameters in the name= and age= accessor methods in a more meaningful way. Let's change the validation code in our name= and age= methods to Ruby's built-in raise method to report any errors.

raise "Something bad happened!"

That's raise as in "raise an issue". Your program is bringing a problem to your attention.

You call raise with a string describing what's wrong. When Ruby encounters the call, it stops what it's doing, and prints your error message. Since this program doesn't do anything to handle the error, it will exit immediately.

Using "raise" in our attribute writer methods

Since we're using raise in both of our writer methods, class Dog we don't need to use an else clause on the if statements. If attr reader :name, :age the new value is invalid and the If "value" is invalid ... def name= (value) raise statement is executed, the →if value == "" program will halt. The statement -->raise "Name can't be blank!" ... execution will halt here. that assigns to the instance @name = value This statement won't be variable will never be reached. reached if "raise" is called. end If "value" is invalid... def age=(value) >if value < 0 raise "An age of #{value} isn't valid!" ... execution will halt here. end reached if "raise" is called. end def report age puts "#{@name} is #{@age} years old." end end Now, if a blank name is passed in to name=, Ruby will report an error, and anonymous = Dog.new anonymous.name = "" Errorthe entire program will exit. in `name=': Name can't be blank! (RuntimeError) You'll get another error message joey = Dog.new if someone tries to set the age to a joey.age = -1Error_ number less than zero. in `age=': An age of -1 isn't valid! (RuntimeError) In a later chapter, we'll see that errors can also be handled by other parts of your program, so that it can continue running. But for now, naughty developers that try to give your Dog instance a blank name or a negative age will know immediately that they have to re-write their code. 0 Awesome! Now, if there's an error in a 0 developer's code, it'll be brought to their attention before a user sees it. Nice work! 71

Our complete Dog class

Here's a file with our complete Dog class, plus some code to create a Dog instance.

```
Sets up "name" and
class Dog
                                  "age" attribute
  attr reader :name, :age _____ reader methods
  def name= (value) Attribute writer
if value == "" method for "@name".
      raise "Name can't be blank!"
    end Data validation.
    @name = value
  end
  def age=(value) 

method for "@age".
    if value < 0
      raise "An age of #{value} isn't valid!"
    end Data validation.
    @age = value
  end
  def move (destination) - Instance method.
    puts "#{@name} runs to the #{destination}."
  end
                Using an instance variable.
  def talk [Instance method.]
    puts "#{@name} says Bark!"
               Using an instance variable.
  end
  def report age method.
    puts "#{@name} is #{@age} years old."
  end
                              – Using instance variables
end
                  ____ Create a new Dog instance.
dog = Dog.new
dog.name = "Daisy" [nitialize attributes.
dog.age = 3 ←
dog.report age
dog.talk Call instance methods.
dog.move("bed") Call instance methods.
                    Do this!
```



We have instance methods that act as *attribute accessors*, letting us get and set the contents of our instance variables.

Daisy

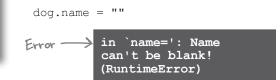
```
puts dog.name
dog.age = 3
puts dog.age
```

We have instance methods that let our dog object do things, like move, make noise, and report its age. The instance methods can make use of the data in the object's instance variables.

```
dog.report_age
dog.talk
dog.move("bed")
```

Daisy is 3 years old. Daisy says Bark! Daisy runs to the bed.

And we've set up our attribute writer methods to *validate* the data passed to them, raising an error if the values are invalid.



Type the above code into a file named "dog.rb". Try adding more Dog instances! Then run ruby dog.rb from a command line. Now, we just need to do the same for the Cat and Bird classes!

Not excited by the prospect of duplicating all that code? Don't worry! The next chapter is all about inheritance, which will make the task easy!



Your Ruby Toolbox

That's it for Chapter 2! You've added methods and classes to your tool box.

Statements Con Methods is m Method parameters can be made optional by providing default values. encl It's legal for a method name to 200 a cc end in ?, !, or =. Methods return the value of their last expression to their caller. You can also specify a method's return value with a return statement.

Classes

A class is a template for creating object instances. An object's class defines its instance methods (what it DOES). Within instance methods, you can create instance variables (what the object KNOWS about itself).

BULLET POINTS

- A method body consists of one or more Ruby statements that will be executed when the method is called.
- Parenthesis should be left off of a method definition if (and only if) you're not defining any parameters.
- If you don't specify a return value, methods will return the value of the last expression evaluated.
- Method definitions that appear within a class definition are treated as instance methods for that class.
- Outside a class definition, instance variables can only be accessed via accessor methods.
- You can call the attr writer, attr reader, and attr accessor methods within your class definition as a shortcut for defining accessor methods.
- Accessor methods can be used to ensure data is valid before it's stored in instance variables.
- The raise method can be called to report an error in your program.

page goal header

3 inheritance

Relying on Your Parents

My siblings and I used to quarrel over our inheritance. But now that we've learned how to share everything, things are working out great!



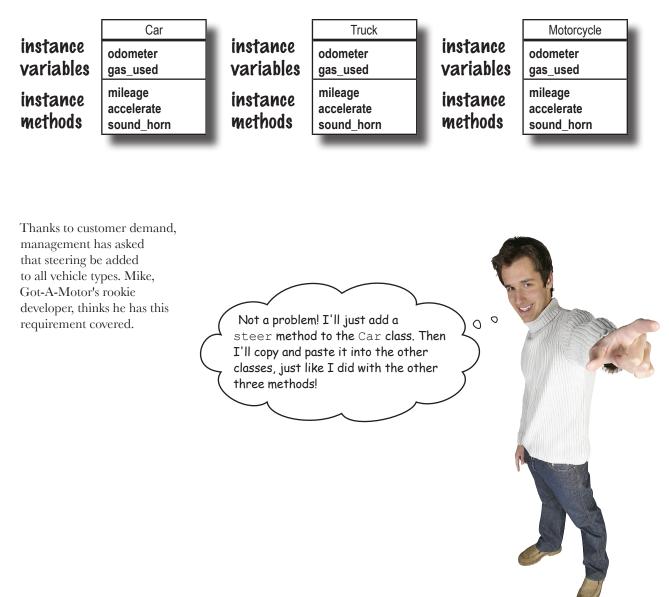
So much repetition! Your new classes representing the different types of vehicles and animals are awesome, it's true. But you're having to *copy instance methods from class to class*. And the copies are starting to fall out of sync - some are fine, and others have bugs. Weren't classes supposed to make code *easier* to maintain?

In this chapter, we'll learn how to use **inheritance** to let your classes *share* methods. Fewer copies means fewer maintenance headaches!

Copy, paste... Such a waste...

Back at Got-A-Motor, Inc., the development team wants to try this "object-oriented programming" thing out for themselves. They've converted their old virtual test drive app to use classes for each vehicle type. They have classes representing cars, trucks, and motorcycles.

Here's what their class structure looks like right now:



Mike's code for the Virtual Test Drive classes

class Car

```
attr_accessor :odometer
attr_accessor :gas_used
```

```
def mileage
  @odometer / @gas_used
end
```

```
def accelerate
   puts "Floor it!"
end
```

```
def sound_horn
   puts "Beep! Beep!"
end
def steer
   puts "Turn front 2 wheels."
```

end

end

```
class Motorcycle
```

```
attr_accessor :odometer
attr_accessor :gas_used
```

```
def mileage
  @odometer / @gas_used
end
```

```
def accelerate
   puts "Floor it!"
end
```

```
def sound_horn
   puts "Beep! Beep!"
end
```

```
def steer <-- Paste!
puts "Turn front 2 wheels."
end
```

end

Marcy is right; this is a maintenance nightmare waiting to happen. First, let's figure out how to address the duplication. Then we'll fix the steer instance method for Motorcycle objects.

class Truck

```
attr_accessor :odometer
attr_accessor :gas_used
```

```
def mileage
  @odometer / @gas_used
end
```

```
def accelerate
   puts "Floor it!"
end
```

```
def sound_horn
   puts "Beep! Beep!"
end
def steer Paste!
   puts "Turn front 2 wheels."
end
```

0

0

end

But Marcy, the team's experienced object-oriented developer, has some reservations about this approach.

> This copy-pasting is a bad idea. What if we needed to change a method? We'd have to change it in every class! And look at the Motorcycle class motorcycles don't have two front wheels!

Inheritance to the rescue!

Fortunately, like most object-oriented languages, Ruby has the concept of **inheritance**, which allows classes to inherit methods from one another. If one class has some functionality, classes that inherit from it can get that functionality *automatically*.

Instead of repeating method definitions across many similar classes, inheritance lets you move the common methods to a single class. You can then specify that other classes inherit from this class. The class with the common methods is referred to as the **superclass**, and the classes that inherit those methods are known as **subclasses**.

If a superclass has instance methods, then its subclasses automatically inherit those methods. You can get access to all the methods you need from the superclass, without having to duplicate the methods' code in each subclass.

Here's how we might use inheritance to get rid of the repetition in the virtual test drive app...



2

We see that the Car, Truck, and Motorcycle classes have several instance methods and attributes in common.



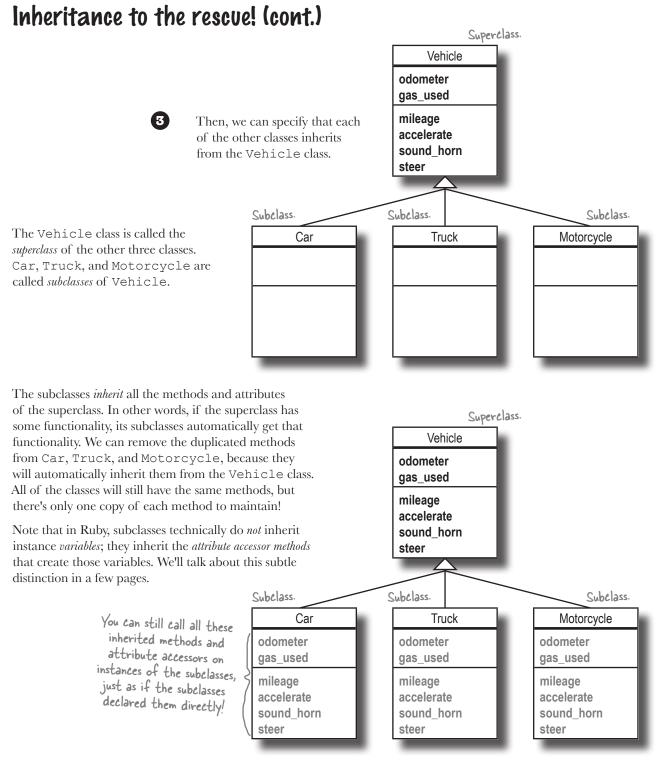
odometer gas_used mileage accelerate sound_horn steer	Truck
accelerate sound_horn	
	accelerate sound_horn

Motorcycle
odometer gas_used
mileage accelerate sound_horn steer
1

. .

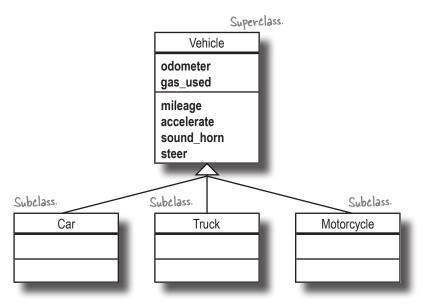
Each one of these classes is a type of vehicle. So we can create a new class, which we'll choose to call Vehicle, and move the common methods and attributes there.

Vehicle
odometer gas_used
mileage
accelerate
sound_horn
steer



Defining a superclass (requires nothing special)

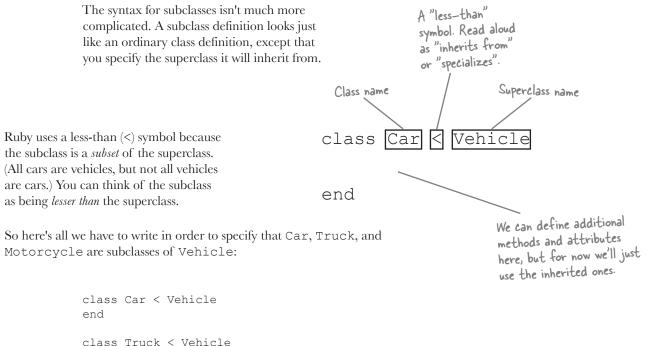
To eliminate the repeated methods and attributes in our Car, Truck, and Motorcycle classes, Marcy has created this design. It moves the shared methods and attributes to a Vehicle *superclass*. Car, Truck, and Motorcycle are all *subclasses* of Vehicle, and they *inherit* all of Vehicle's methods.



There's actually no special syntax to define a superclass in Ruby; it's just an ordinary class. (Most object-oriented languages are like this.) class Vehicle

```
All attributes will be (attr_accessor :odometer
inherited when we (attr_accessor :gas_used
  declare a subclass.
                   def accelerate
                      puts "Floor it!"
                   end
                   def sound horn
                      puts "Beep! Beep!"
       So will all
                   end
        instance
        methods.
                   def steer
                      puts "Turn front 2 wheels."
                   end
                   def mileage
                      return @odometer / @gas used
                   end
```

Defining a subclass (is really easy)



end

class Motorcycle < Vehicle
end</pre>

As soon as you define them as subclasses, Car, Truck, and Motorcycle inherit all the attributes and instance methods of Vehicle. Even though the subclasses don't contain any code of their own, any instances we create will have access to all of the superclass's functionality!

```
truck = Truck.new
truck.accelerate
truck.steer
car = Car.new
car.odometer = 11432
car.gas_used = 366
puts "Lifetime MPG:"
puts car.mileage
```

Floor it! Turn front 2 wheels. Lifetime MPG: 31

Our Car, Truck, and Motorcycle classes have all the same functionality they used to, without all the duplicated code. Using inheritance will save us a lot of maintenance headaches!

Adding methods to subclasses

As it stands, there's no difference between our Truck class and the Car or Motorcycle classes. But what good is a truck, if not for hauling cargo? Got-A-Motor wants to add a load_bed method for Truck instances, as well as a cargo attribute to access the bed contents.

It won't do to add cargo and load_bed to the Vehicle class, though. The Truck class would inherit them, yes, but so would Car and Motorcycle. Cars and motorcycles don't *have* cargo beds!

So instead, we can define a cargo attribute and a load_bed method *directly on the* Truck *class*.

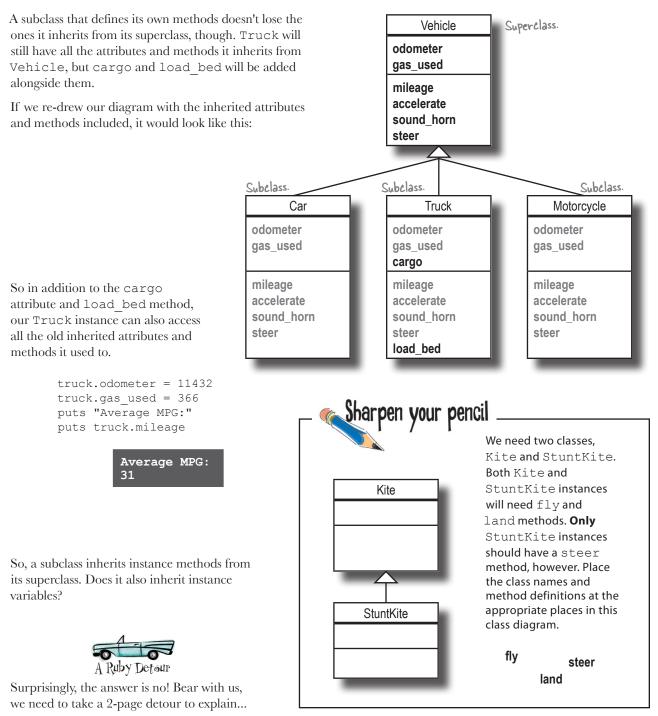
class Truck < Vehicle
 attr_accessor :cargo
 def load_bed(contents)
 puts "Securing #{contents} in the truck bed."
 @cargo = contents
 end</pre>

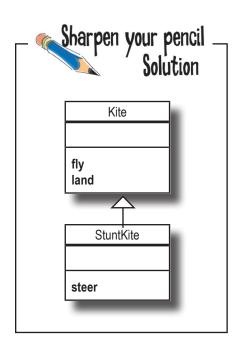
end

Superclass. Vehicle If we were to draw the diagram of Vehicle and its subclasses again now, it would look like this: odometer gas used mileage accelerate sound horn steer Subclass. Subclass. Subclass. Car Truck Motorcycle cargo load bed With these code changes in place, we can create a new Truck instance, then load and access its cargo. truck = Truck.new truck.load bed("259 bouncy balls") puts "The truck is carrying #{truck.cargo}."

> Securing 259 bouncy balls in the truck bed. The truck is carrying 259 bouncy balls.

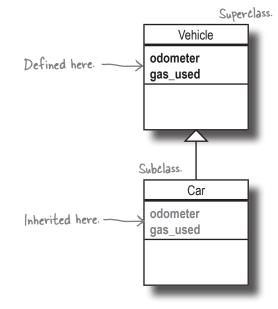
Subclasses keep inherited methods alongside new ones





Instance variables are NOT inherited! A Ruby Detour

It's easy to form the (incorrect) impression that instance variables are inherited from the superclass. Let's take another look at our class diagram, focusing on the attributes of the Vehicle and Car classes...



All Ruby objects have a method called instance_variables that we can call to see what instance variables are defined for that object. So if we create a new Car and assign values to its odometer and gas_used attributes...

```
car = Car.new
car.odometer = 22914
car.gas_used = 728
```

...then call the instance_variables method to see what instance variables it has...

puts car.instance_variables



...it sure *looks* like the <code>@odometer</code> and <code>@gas_used</code> instance variables got inherited from the <code>Vehicle</code> superclass.

But what *actually* gets inherited are the odometer and gas_used *instance methods* (the attribute accessor methods). These methods just *happen* to assign to instance variables named @odometer and @gas_used (because that's the Ruby convention). The variables are *created* on the car object *at the time a value is assigned to them*.

The only thing that Ruby subclasses ever inherit are instance methods. Instance variables usually come along for the ride, though.

Instance variables are NOT inherted! (cont.) A Ruby Detour

To prove that it's the odometer and gas used instance *methods* that are inherited from Vehicle, and not the Odometer and Ogas used instance variables, let's try breaking the convention. We'll override the Car subclass's attribute accessor methods to write to and read from totally different instance variables.

```
class Car < Vehicle
 def odometer=(new value)
    @banana = new value
 end
 def odometer
    @banana
 end
 def gas used=(new value)
    @apple = new value
 end
 def gas used
    @apple
 end
end
```

Now, we can run the very same code to create a Car instance:

car = Car.new car.odometer = 22914car.gas used = 728

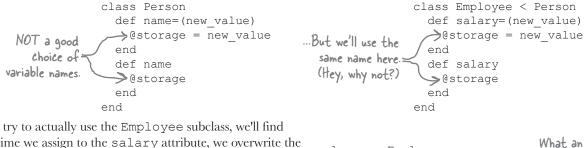
...But the odometer= and gas used= methods will assign to different instance variables:

puts car.instance variables

Note the complete -absence of Godometer banana @apple and Qgas used!

End of Ruby Detour

So, why worry about the fact that instance variables aren't inherited? As long as you follow the convention of ensuring your instance variable names match your accessor method names, you won't have to. But if you deviate from that convention, look out! You may find that a subclass can interfere with it's superclass's functionality by overwriting its instance variables.



When we try to actually use the Employee subclass, we'll find that any time we assign to the salary attribute, we overwrite the name attribute, because both are using the *same* instance variable.

What's the lesson here? Ensure you're using sensible variable names that match your attribute accessor names. That simple practice should be enough to keep you out of trouble!



Overriding methods

00

Marcy, the team's experienced objectoriented developer, has re-written our Car, Truck, and Motorcycle classes as subclasses of Vehicle. They don't need any methods or attributes of their own - they inherit everything from the superclass! But Mike points out an issue with this design...

Pretty slick, Marcy. But you forgot one little detail: the Motorcycle class needs a specialized steer method!

motorcycle = Motorcycle.new
motorcycle.steer

Turn front 2 wheels

Not a problem - I can just **override** that method for Motorcycle! One wheel too many, for a motorcycle!

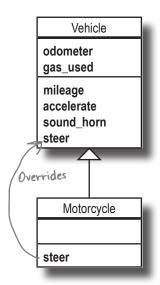
If the superclass's behavior isn't what you need in the subclass, inheritance gives you another mechanism to help: method *overriding*. When you **override** one or more methods in a subclass, you replace the inherited methods from the superclass with methods specific to the subclass.

class Motorcycle < Vehicle
 def steer
 puts "Turn front wheel."
 end
end</pre>

Now, if we call steer on a Motorcycle instance, we'll get the overriding method. That is, we'll get the version of steer defined within the Motorcycle class, not the version from Vehicle.

motorcycle.steer

Turn front wheel.



00

Overriding methods (continued)

If we call any other methods on a Motorcycle instance, though, we'll get the inherited method.

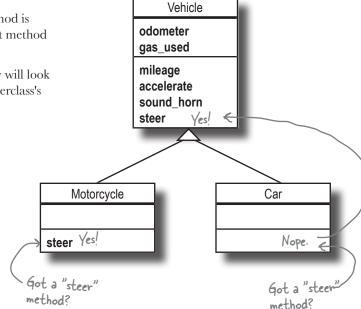
motorcycle.accelerate



How does this work?

If Ruby sees that the requested method is defined on a subclass, it will call that method and stop there.

But if the method's not found, Ruby will look for it on the superclass, then the superclass's superclass, and so on, up the chain.



Everything seems to be working again! When changes are needed, they can be made in the Vehicle class, and they'll propagate to the subclasses automatically, meaning everyone gets the benefit of updates sooner. If a subclass needs specialized behavior, it can simply override the method it inherited from the superclass.

Nice work cleaning up Got-A-Motor's code! Up next, we have a couple exercises where you can practice working with superclasses and subclasses.

Then, we'll take another look at the Fuzzy Friends code. They still have a lot of redundant methods in their application's classes. We'll see if inheritance and method overriding can help them out. Q: Can you have more than one level of inheritance? That is, can a subclass have its own subclasses?

A: Yes! If you need to override methods on some of your subclass's instances, but not others, you might consider making a subclass of the subclass.

```
class Car < Vehicle
end
```

```
class DragRacer < Car
  def accelerate
    puts "Inject nitrous!"
  end
end</pre>
```

Don't overdo it, though! This kind of design can rapidly become very complex. Ruby doesn't place a limit on the number of levels of inheritance, but most Ruby developers don't go more than one or two levels deep.

bumb Questions

Q: You said that if a method is called on an instance of a class and Ruby doesn't find the method, it will look on the superclass, then the superclass's superclass... What happens if it runs out of superclasses without finding the method?

A: After searching the last superclass, Ruby gives up the search. That's when you get one of those "undefined method" errors we've been seeing.

Car.new.fly

undefined method
`fly' for
#<Car:0x007ffec48c>

Q: When designing an inheritance hierarchy, which should I design first, the subclass or the superclass?

A: Either! You might not even realize you need to use inheritance until after you've started coding your application.

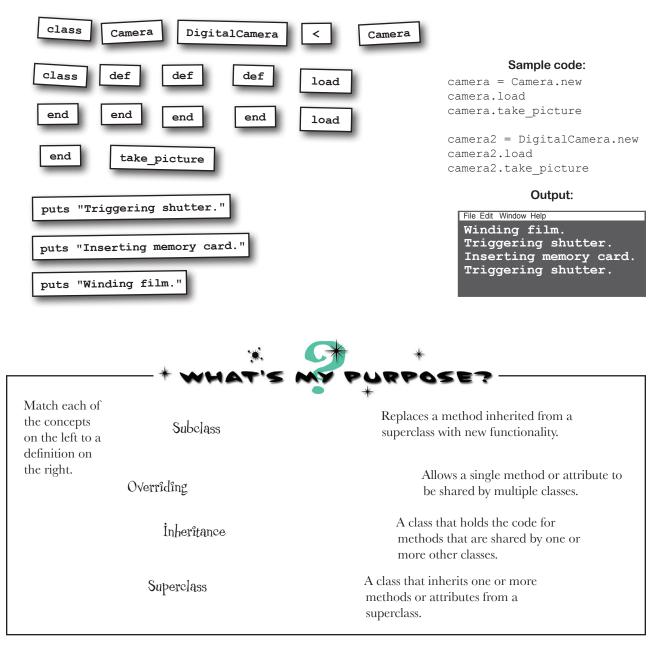
When you discover that two related classes need similar or identical methods, though, just make those classes into subclasses of a new superclass. Then move those shared methods into the superclass. There: you've designed the subclasses first.

Likewise, when you discover that only some instances of a class are using a method, create a new subclass of the existing class, and move the method there. You've just designed the superclass first!



Code Magnets

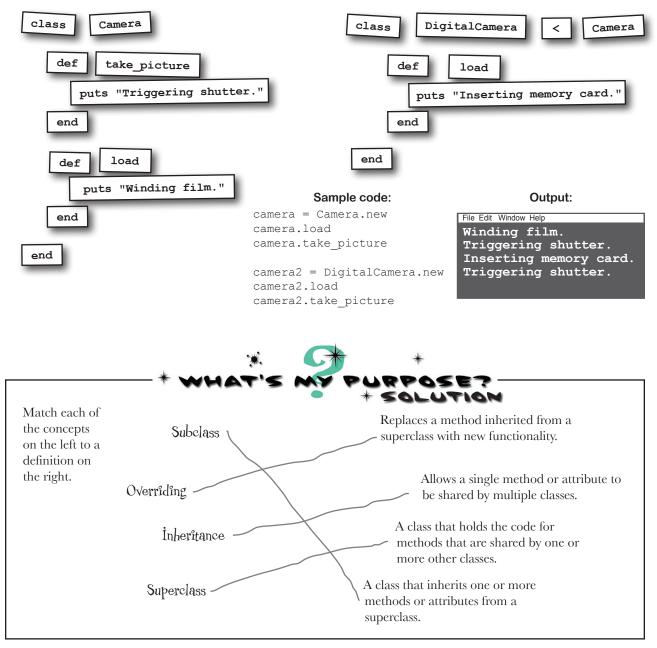
A Ruby program is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working superclass and subclass, so the sample code below can execute and produce the given output?





Code Magnets Solution

A Ruby program is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working superclass and subclass, so the sample code below can execute and produce the given output?



Bringing our animal classes up to date with inheritance

Remember the Fuzzy Friends virtual storybook application from last chapter? We did a lot of excellent work on the Dog class. We added name and age attribute accessor methods (with validation), and updated the talk, move, and report_age methods to use the @name and @age instance variables.

Here's a recap of the code we have so far:

```
Creates methods to
                   class Dog
get current values of
Öname and @age. -
                    → attr reader :name, :age
                       def name=(value)
                         if value == ""
                           raise "Name can't be blank!"
   We create our own
                         end
   attribute writer
                         @name = value
   methods, so we can
                       end
   check that the new <
   values are valid.
                       def age=(value)
                         if value < 0
                           raise "An age of #{value} isn't valid!"
                         end
                         @age = value
                       end
                       def talk
                       puts "#{@name} says Bark!"
                       end
       Other instance
                       def move (destination)
       methods for our <
                       puts "#{@name} runs to the #{destination}."
       Dog objects.
                       end
                       def report age
                         puts "#{@name} is #{@age} years old."
                       end
                     end
```

The Bird and Cat classes have been completely left behind, however, even though they need almost identical functionality.

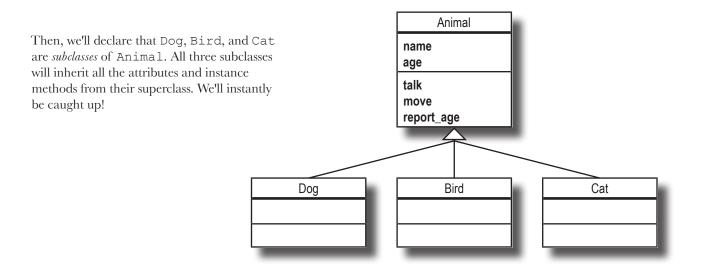
Let's use this new concept of inheritance to create a design that will bring all our classes up to date at once (and keep them updated in the future).

Designing the animal class hierarchy

We've added lots of new functionality to our Dog class, and now we want it in the Cat and Bird classes as well...

We want all the classes to have name and age attributes, as well as talk, move, and report_age methods. Let's move all of these attributes and methods up to a new class, which we'll call Animal.

Animal	
name	
age	
talk	
move	
report_age	
_	



Code for the Animal class and its subclasses

class Animal Here's code for the Animal superclass, with all the old methods attr reader :name, :age from Dog moved into it... def name=(value) if value == "" raise "Name can't be blank!" end @name = value end def age=(value) if value < 0 raise "An age of #{value} isn't valid!" end @age = value The exact same end code that was def talk in the Dog class! puts "#{@name} says Bark!" end def move(destination) puts "#{@name} runs to the #{destination}." end def report age puts "#{@name} is #{@age} years old." end end We dont' have to write any And here are the other classes, class Dog < Animal methods here; these classes will end rewritten as subclasses of Animal. inherit all the methods from class Bird < Animal the Animal class above! end class Cat < Animal end

Overriding a method in the Animal subclasses

With our Dog, Bird, and Cat classes re-written as subclasses of Animal, they don't need any methods or attributes of their own - they inherit everything from the superclass!

```
whiskers = Cat.new("Whiskers")
fido = Dog.new("Fido")
polly = Bird.new("Polly")
polly.age = 2
polly.report_age
fido.move("yard")
whiskers.talk
```

Polly is 2 years old. Fido runs to the yard. Whiskers says Bark!

- Wait... Whiskers is a Cat...

Looks good, except for one problem ... our Cat instance is barking.

The subclasses inherited this method from Animal:

```
def talk
  puts "#{@name} says Bark!"
end
```

That's appropriate behavior for a Dog, but not so much for a Cat or a Bird.

```
whiskers = Cat.new("Whiskers")
polly = Bird.new("Polly")
whiskers.talk
polly.talk
Whiskers says Bark!
```

This code will override the talk method that was inherited from Animal:

Now, when you call talk on Cat or Bird instances, you'll get the overridden methods.

```
whiskers.talk Whiskers says Meow!
polly.talk Polly says Chirp! Chirp!
```

Animal

name

age

talk

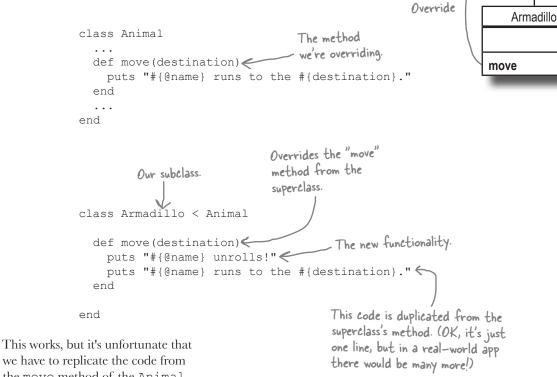
move

report age

We need to get at the overridden method!

Next up, Fuzzy Friends wants to add armadillos to their interactive storybook. (Yeah, the little anteater-like critters that can roll into an armored ball to protect themselves from predators and overly-playful dogs.) We can simply add Armadillo as a subclass of Animal.

There's a catch, though; before they can run anywhere, they have to unroll. The move method will have to be overridden to reflect this fact.

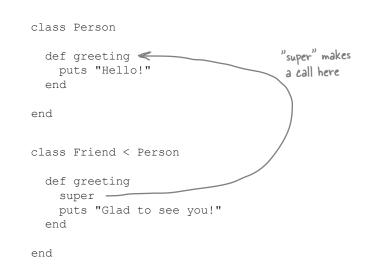


we have to replicate the code from the move method of the Animal class.

What if we could override the move method with new code, and still harness the code from the superclass? Ruby has a mechanism to do just that...

The "super" keyword

When you use the super keyword within a method, it makes a call to a method of the same name on the superclass.



If we make a call to the *overriding* method on the *subclass*, we'll see that the super keyword makes a call to the *overridden* method on the *superclass*:

Friend.new.greeting

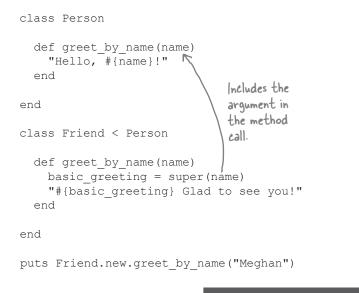
Hello! Glad to see you!

The super keyword works like an ordinary method call in almost every respect.

For example, the superclass method's return value becomes the value of class Person the super expression: The method return value. def greeting "Hello!" end end class Friend < Person Assigns "Hello!" to def greeting basic greeting basic greeting = super < "#{basic greeting} Glad to see you!" end end Hello! Glad to see you! puts Friend.new.greeting

The "super" keyword (continued)

Another way in which using the super keyword is like a regular method call: you can pass it arguments, and those arguments will be passed to the superclass's method.



Hello, Meghan! Glad to see you!

But here's a way that super *differs* from a regular method call: if you leave the arguments *off*, the superclass method will automatically be called with the same arguments that were passed to the subclass method.



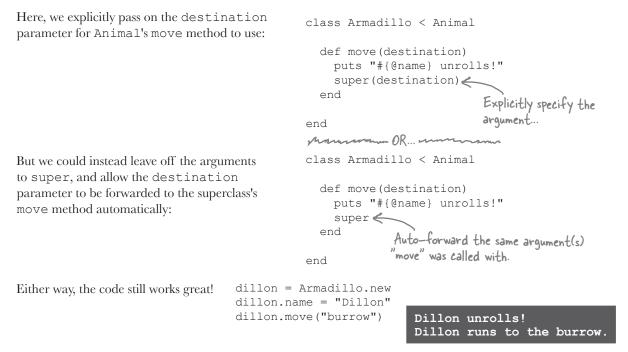
A super-powered subclass

Now, let's use our new understanding of super to eliminate a little duplicated code from the move method in our Armadillo class.

```
class Animal
Here's the method we're inheriting from the
Animal superclass:
                                                 def move (destination)
                                                   puts "#{@name} runs to the #{destination}."
                                                 end
                                                 . . .
                                               end
                                                                                   Here's that
                                                                                   duplicated line.
And here's the overridden version in the
                                               class Armadillo < Animal
Armadillo subclass:
                                                 def move (destination)
                                                   puts "#{@name} unrolls!"
                                                   puts "#{@name} runs to the #{destination}."
                                                 end
```

end

We can replace the duplicated code in the subclass's move method with a call to super, and rely on the superclass's move method to provide that functionality.



Your mastery of class inheritance has wrung the repetition out of your code like water from a sponge. And your co-workers will thank you - less code means less bugs! Great job!



Below you'll find code for three Ruby classes. The code snippets on the right use those classes, either directly or through inheritance. Fill in the blanks below each snippet with what you think its output will be. Don't forget to take method overriding and the "super" keyword into account! (We've filled in the first one for you.)

```
class Robot
```

```
attr_accessor :name
def activate
  puts "#{@name} is powering up"
end
```

```
def move(destination)
  puts "#{@name} walks to #{destination}"
end
```

```
end
```

```
class TankBot < Robot
```

```
attr accessor :weapon
```

```
def attack
   puts "#{@name} fires #{@weapon}"
end
```

```
def move(destination)
  puts "#{@name} rolls to #{destination}"
end
```

```
end
```

```
class SolarBot < Robot
```

```
def activate
   puts "#{@name} deploys solar panel"
   super
end
```

```
end
```

Your answers:

```
tank = TankBot.new
tank.name = "Hugo"
tank.weapon = "laser"
tank.activate
tank.move("test dummy")
tank.attack
```

```
Hugo is powering up
sunny = SolarBot.new
sunny.name = "Sunny"
sunny.activate
sunny.move("tanning bed")
```



Below you'll find code for three Ruby classes. The code snippets on the right use those classes, either directly or through inheritance. Fill in the blanks below each snippet with what you think its output will be. Don't forget to take method overriding and the "super" keyword into account!

```
class Robot
  attr accessor :name
  def activate
   puts "#{@name} is powering up"
  end
  def move(destination)
   puts "#{@name} walks to #{destination}"
  end
end
class TankBot < Robot
  attr accessor :weapon
  def attack
   puts "#{@name} fires #{@weapon}"
  end
  def move(destination)
   puts "#{@name} rolls to #{destination}"
  end
end
class SolarBot < Robot
  def activate
   puts "#{@name} deploys solar panel"
    super
  end
end
```

tank = TankBot.new
tank.name = "Hugo"
tank.weapon = "laser"
tank.activate
tank.move("test dummy")
tank.attack

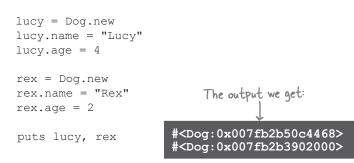
```
Hugo is powering up
Hugo rolls to test dummy
Hugo fires laser
```

sunny = SolarBot.new
sunny.name = "Sunny"
sunny.activate
sunny.move("tanning bed")

```
Sunny deploys solar panel
Sunny is powering up
Sunny walks to tanning bed
```

Difficulties displaying Dogs

Let's make one more improvement to our Dog class, before we declare it finished. Right now, if we pass a Dog instance to the print or puts methods, the output isn't too useful:



We can tell that they're Dog objects, but beyond that it's very hard to tell one Dog from another. It would be far nicer if we got output like this:



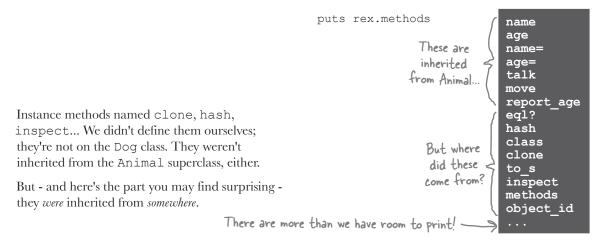
When you pass an object to the puts method, Ruby calls the to_s instance method on it to convert it to a string for printing. We can call to_s explicitly, and get the same result:

puts lucy.to_s, rex.to_s

#<Dog:0x007fb2b50c4468> #<Dog:0x007fb2b3902000>

Now, here's a question: where did that to s instance method come from?

Indeed, where did *most* of these instance methods on Dog objects come from? If you call the method named methods on a Dog instance, only the first few instance methods will look familiar...

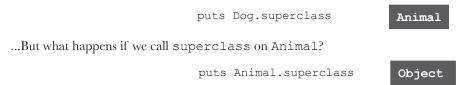


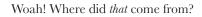
The Object class

Where could our Dog instances have inherited all these instance methods from? We don't define them in the Animal superclass. And we didn't specify a superclass for Animal...

> class Dog < Animal The superclass for end Dog is Animal. class Animal end No superclass specified!

Ruby classes have a superclass method that you can call to get their superclass. The result of using it on Dog isn't suprising:





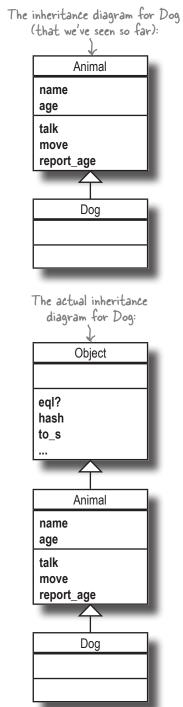
When you define a new class, Ruby implicitly sets a class called Object as its superclass (unless you specify a superclass yourself).

So writing this:

class Animal ... end

... is equivalent to writing this:

class Animal < Object
 ...
end</pre>



Why everything inherits from the Object class

If you don't explicitly specify a superclass for a class you define, Ruby implicitly sets a class named Object as the superclass...

Even if you *do* specify a superclass for your class, that superclass probably inherits from Object. That means almost every Ruby object, directly or indirectly, has Object as a superclass!

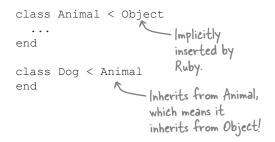
Ruby does this because the Object class defines dozens of useful methods that almost all Ruby objects need. This includes a lot of the methods that we've been calling on objects so far:

- The to s method converts an object to a string for printing
- The inspect method converts an object to a debug string
- The class method tells you which class an object is an instance of
- The methods method tells you what instance methods an object has
- The instance variables method gives you a list of an object's instance variables

...And there are many others. The methods inherited from the Object class are fundamental to the way Ruby works with objects.

We hope you've found this little tangent informative, but it doesn't help us with our original problem: our Dog objects are still printing in a gibberish format.

Or does it?



Overriding the inherited method

We specified that the superclass of the Dog class is the Animal class. And we learned that because we *didn't* specify a superclass for Animal, Ruby automatically set the Object class as its superclass.

That means that Animal instances inherit a to_s method from Object. Dog instances, in turn, inherit to_s from Animal. When we pass a Dog object to puts or print, its to_s method is called, to convert it to a string.

Do you see where we're headed? If the to_s method is the source of the gibberish strings being printed for Dog instances, and to_s is an *inherited* method, all we have to do is *override* to_s *on the* Dog *class*!

```
class Dog < Animal

def to_s

"#{@name} the dog, age #{age}" 
This return value is the

end
format we'd like to see.
```

end

Are you ready? Let's try it.

```
lucy = Dog.new
lucy.name = "Lucy"
lucy.age = 4
rex = Dog.new
rex.name = "Rex"
rex.age = 2
puts lucy.to s, rex.to s
```

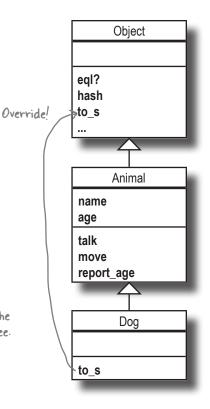
Lucy the dog, age 4 Rex the dog, age 2

It works! No more "#<Dog:0x007fb2b50c4468>". This is actually readable!

One more tweak: the to_s method is already called when printing objects, so we can leave that off:

puts lucy, rex

This new output format will make debugging the virtual storybook much easier. And you've gained a key insight into how Ruby objects work - inheritance plays a vital role!



there are no Dumb Questions

Q: I tried this code in irb
instead of using the ruby
command. After I override to_s,
if I type lucy = Dog.new
into irb, I still see something like
"#<Dog: 0x007fb2b50c4468>".
Why don't I see the dog's name and
age?

A: The values that irb shows you are the result of calling inspect on an object, not to_s. You won't see the results of to_s until you set the name and age, and pass the object to puts.



Your Ruby Toolbox

That's it for Chapter 3! You've added inheritance to your tool box.

Statements Com Methods is m Me Classes encl It's ob Inheritance lets a subclass inherit An methods from a superclass. a cc end call do A subclass can define its own Me kni methods in addition to the methods it inherits. ret A subclass can override inherited sta methods, replacing them with its own version.

BULLET POINTS

- Any ordinary Ruby class can be used as a superclass.
- To define a subclass, simply specify a superclass in the class definition.
- Instance variables are not inherited from a superclass, but the methods that create and access instance variables are inherited.
- The super keyword can be used within a subclass method to call the overridden method of the same name on the superclass.
- If you don't specify arguments to the super keyword, it takes all arguments that the subclass method was called with, and passes them on to the superclass method.
- The expression value of the super keyword is the return value of the superclass method it calls.
- When defining a class, Ruby implicitly sets the Object class as the superclass, unless you specify one.
- Almost every Ruby object has instance methods from the Object class, inherited either directly, or through another superclass.
- The to_s, methods, instance_variables, and class methods are all inherited from the Object class.

page goal header

4 initializing instances

Off to a Great Start



Right now, your class is a time bomb. Every instance you create starts out as a clean slate. If you call certain instance methods before adding data, an error will be raised that will bring your whole program to a screeching halt.

We're going to show you a couple ways to create objects that are safe to use right away. We'll start with the initialize method, which lets you pass in a bunch of arguments to set up an object's data *at the time you create it*. Then we'll show you how to write **class methods**, which you can use to create and set up an object even **more** easily.

Payroll at Chargemore

You've been tasked with creating a payroll system for Chargemore, a new chain of department stores. They need a system that will print pay stubs for their employees.

Chargemore employees are paid for two-week pay periods. Some employees are paid a two-week portion of their annual salary, and some are paid for the number of hours they work within a two-week period. For starters, though, we're just going to focus on the salaried employees.

A pay stub needs to include the following information:

- The employee name
- The amount of pay an employee received during a two-week pay period

So... here's what the system will need to *know* for each employee:

- Employee name
- Employee salary

And here's what it will need to *do*:

• Calculate and print pay for a two-week period

This sounds like the ideal place to create an Employee class! Let's try it, using the same techniques that we covered back in Chapter 2.

We'll set up attribute reader methods for @name and @salary instance variables, then add writer methods (with validation). Then we'll add a print_pay_stub instance method that prints the employee's name, and their pay for the period.

Employee
name salary
print_pay_stub



@name = "Kara Byrd" @salary = 45000

@name = "Ben Weber" @salary = 50000

@name = "Amy Blake" @salary = 50000



An Employee class

Here's some code to implement our Employee class ...

```
We need to create attribute
                                      writer methods manually, so
                                      we can validate the data. We
                                      can create reader methods
class Employee
                                     _ automatically, though.
 attr reader :name, :salary <
                                             Report an error if the
 def name=(name)
                                        name is blank.
    if name == ""
      raise "Name can't be blank!"
    end
    Qname = name - Store the name in an
                        instance variable.
 end
 def salary=(salary)
    if salary < 0
      salary is negative.
   end
   Osalary = salary - Store the salary in an
 end
                            instance variable.
   puts "Name: #{@name}" Print the employee name.

pay_for_period = (@salary / 365) * 14 Grant of the employee's salary.
 def print pay stub
    puts "Pay This Period: $#{pay_for_period}" <---
  end
                                                     Print the amount paid.
end
```

(Yes, we realize that this doesn't account for leap years and holidays and a host of other things that real payroll apps must consider. But we wanted a print pay stub method that fits on one page.)

Creating new Employee instances

Now that we've defined an Employee class, we can create new instances, and assign to their name and salary attributes.

```
amy = Employee.new
amy.name = "Amy Blake"
amy.salary = 50000
```

Thanks to validation code in our name= method, we have protection against the accidental assignment of blank names.



Our salary= method has validation to ensure negative numbers aren't assigned as a salary.

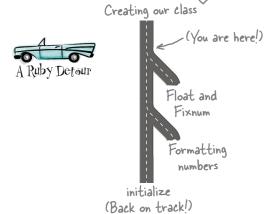


And when an Employee instance is properly set up, we can use the stored name and salary to print a summary of the employee's pay period.

Hmmm... It's typical to display two decimal places when showing currency, though. And did that calculation really come out to an even dollar amount?

Before we go on to perfect our Employee class, it looks like we have a bug to fix. And that will require us to go on a couple brief detours. (But you'll learn some number formatting skills that you'll need later, promise!)

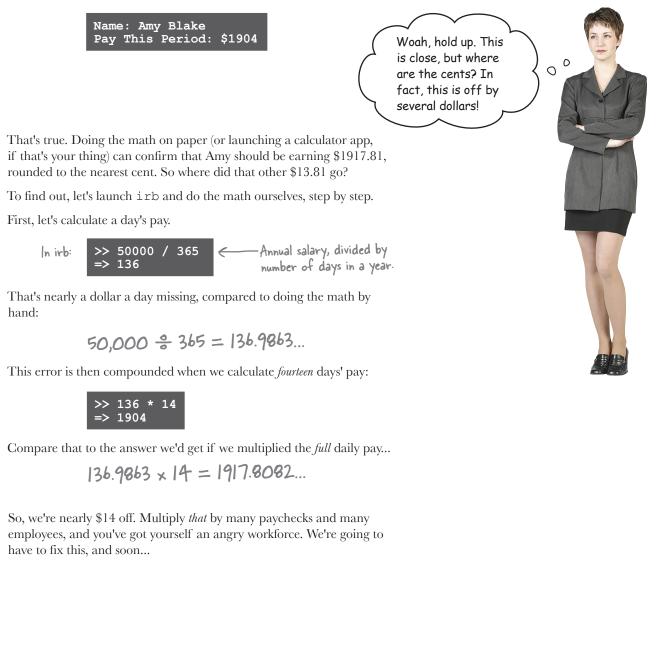
- Our employee pay is getting its decimal places chopped off. To fix this, we'll need to look at the difference between Ruby's Float and Fixnum numeric classes.
- 2. We don't want to display too *many* decimal places, either, so we'll need to look at the format method to format our numbers properly.





A division problem

We're working to make the perfect Employee class to help us calculate payroll for the Chargemore department store. But there's a little detail we have to take care of, first...



A Ruby Detour Division with Ruby's Fixnum class

The result of our Ruby expression to calculate 2 weeks of an employee's pay doesn't match up with doing the math by hand...

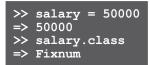
```
>> 50000 / 365 * 14
=> 1904 50,000 ÷ 365 × 14 = 1917.8082...
```

The problem here is that when dividing instances of the Fixnum class (a Ruby class that represents integers), Ruby rounds fractional numbers *down* to the nearest whole number.



It rounds the number because Fixnum instances aren't *meant* to store numbers with decimal places. They're intended for use in places where only whole numbers make sense, like counting employees in a department or the number of items in a shopping cart. When you create a Fixnum, you're telling Ruby: "I expect to only be working with whole numbers here. If anyone does math with you that results in a fraction, I want you to throw those pesky decimal places away."

How can we know whether we're working with Fixnums? We can call the class instance method on them. (Remember we talked about the Object class back in Chapter 3? The class method is one of the instance methods inherited from Object.)



Or, if you'd rather save yourself the trouble, just remember that any number in your code that *doesn't* have a decimal point in it will be treated as a Fixnum by Ruby.

Any number in your code that *does* have a decimal point in it gets treated as a Float (the Ruby class that represents floating-point decimal numbers):

>>	salary = 50000.0
=>	50000.0
>>	salary.class
=>	Float

If it's got a decimal point, it's a Float. If it doesn't, it's a Fixnum. 273 273.4

Fixnum Float

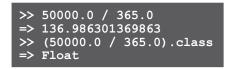


Division with Ruby's Float class

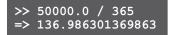
We loaded up irb, and saw that if we divide one Fixnum (integer) instance by another Fixnum, Ruby rounds the result *down*.

>> 50000 / 365 => 136 Should be 136.9863...-

The solution, then, is to use Float instances in the operation, which we can get by including a decimal point in our numbers. If you do, Ruby will give you a Float instance back:



It doesn't even matter whether both the dividend and divisor are Float instances; Ruby will give you a Float back as long as *either* operand is a Float.



It holds true for addition, subtraction, and multiplication as well: Ruby will give you a Float if *either* operand is a Float:

>>	50000 +	1.5
=>	50001.5	
>>	50000 -	1.5
=>	49998.5	
>>	50000 *	1.5
=>	75000.0	

When the first operand is a	And the second operand is a	The result is a
Fixnum	Fixnum	Fixnum
Fixnum	Float	Float
Float	Fixnum	Float
Float	Float	Float

And of course, with addition, subtraction, and multiplication, it doesn't matter whether both operands are Fixnum instances, because there's no fractional number to lose in the result. The only operation where it really matters is division. So, remember this rule:

When doing division, make sure at least one operand is a Float.

Let's see if we can use this hard-won knowledge to fix our Employee class.

A Ruby Detour Fixing the salary rounding error in Employee

As long as one of the operands is a Float, Ruby won't truncate the decimals from our division operation.

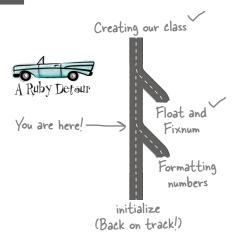
>> 50000 / 365.0 => 136.986301369863

With this rule in mind, we can revise our Employee class to stop truncating the decimals from employees' pay:

Now we have a new problem, though: look what happens to the output!

Name: Jane Doe Pay This Period: \$1917.8082191780823

We're showing a little *too much* precision! Currency is generally expected to be shown with just two decimal places, after all. So, before we can go back to building the perfect Employee class, we need to go on one more detour...





Formatting Numbers for Printing

Our print_pay_stub method is displaying too many decimal places. We need to figure out how to round the displayed pay to the nearest penny (2 decimal places).

Name: Jane Doe	
Pay This Period:	\$1917.8082191780823

To deal with these sort of formatting issues, Ruby provides the format method.

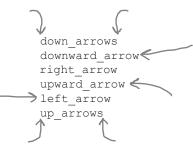
Here's a sample of what this method can do. It may look a little confusing, but we'll explain it all on the next few pages!

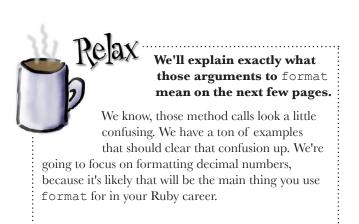
```
result = format("Rounded to two decimal places: %0.2f", 3.14159265)
puts result
```

Rounded to two decimal places: 3.14

So, it looks like format *can* help us limit our displayed employee pay to the correct number of places. The question is, *how*? To be able to use this method effectively, we'll need to learn about two features of format:

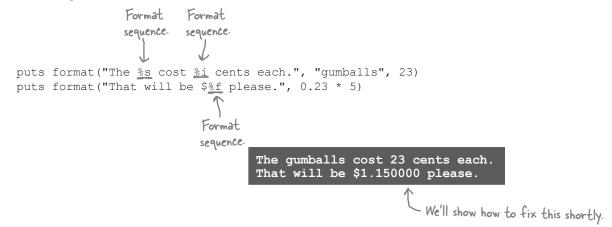
- 1. Format sequences (the little %0.2f above is a format sequence)
- 2. Format sequence widths (that's the 0.2 in the middle of the format sequence)







The first argument to format is a string that will be used to format the output. Most of it is formatted exactly as it appears in the string. Any percent signs (%), however, will be treated as the start of a **format sequence**, a section of the string that will be substituted with a value in a particular format. The remaining arguments are used as values for those format sequences.



Format sequence types

The letter following the percent sign indicates the type of value that's expected. The most common types are:

ĕS	string		
%i	integer	<pre>puts format("A string: %s", "hello")</pre>	A string: hello
	0	puts format("An integer: %i", 15)	An integer: 15
%f	floating-point decimal	puts format("A float: %f", 3.1415)	A float: 3.141500

So %f is for floating-point decimal numbers... We can use that sequence type to format the currency in our pay stubs.

By itself, though, the %f sequence type won't help us. The results still show too many decimal places.

puts format("\$%f", 1917.8082191780823)

\$1917.808219

Up next, we'll look at a fix for that situation: the format sequence width.

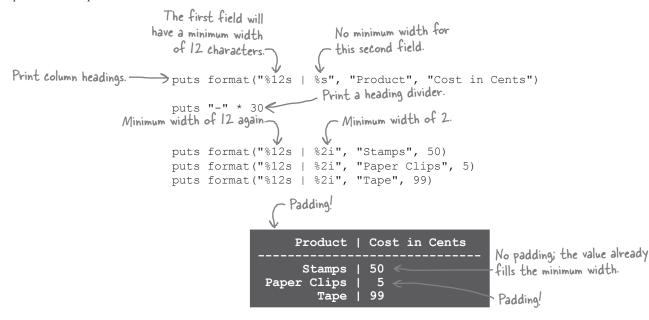


Format sequence width

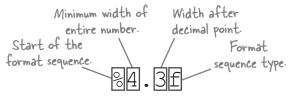
Here's the useful part of format sequences: they let you specify the *width* of the resulting field.

Let's say we want to format some data in a plain-text table. We need to ensure the formatted value fills a minimum number of spaces, so that the columns align properly.

You can specify the minimum width after the percent sign in a format sequence. If the argument for that format sequence is shorter than the minimum width, it will be padded with spaces until the minimum width is reached.



And now we come to the part that's important for today's task: you can use format sequence widths to specify the precision (the number of displayed digits) for floating point numbers. Here's the format:



The minimum width of the entire number includes decimal places. If it's included, shorter numbers will be padded with spaces at the start until this width is reached. If it's omitted, no spaces will ever be added.

The width after the decimal point is the maximum number of digits to show. If a more precise number is given, it will be rounded (up or down) to fit in the given number of decimal places.

A Ruby Detour Format sequence width with floating-point numbers

So when working with floating-point numbers, format sequence widths let us specify the number of digits displayed before *and* after the decimal point. Could this be the key to fixing our pay stubs?

Here's a quick demonstration of various width values in action:

```
def test format (format string)
  print "Testing '#{format string}': "
  puts format (format string, 12.3456)
end
                                                             .Rounded to 3 places.
test format "%7.3f"
                        Testing '%7.3f':
                                              12.346
                                                       \epsilon
                                                             -Rounded to 2 places.
                                               12.35
test format "%7.2f"
                        Testing '%7.2f':
                                                       \epsilon
                                                              Rounded to I place.
test format "%7.1f"
                        Testing '%7.1f':
                                                12.3
                                                       |
                                                              Rounded to I place, no padding
                                  '%.1f': 12.3
                        Testing
test format "%.1f"
                                                             . Rounded to 2 places, no padding.
                        Testing
                                  '%.2f': 12.35
test format "%.2f"
```

That last format, "%.2f", will let us take floating-point numbers of any precision and round them to two decimal places. (It also won't do any unnecessary padding.) This format is ideal for showing currency, and it's just what we need for our print_pay_stub method!

```
puts format("$%.2f", 1917.8082191780823)
puts format("$%.2f", 1150.6849315068494)
puts format("$%.2f", 3068.4931506849316)
$3068.49
```

Previously, our calculated pay for our Employee class's print_pay_stub method was displayed with excess decimal places:

```
salary = 50000
puts "$#{(salary / 365.0) * 14}"
$1917.8082191780823
```

But now, we finally have a format sequence that will round a floating-point number to two decimal places:

puts format("\$%.2f", (salary / 365.0) * 14) \$1917.81

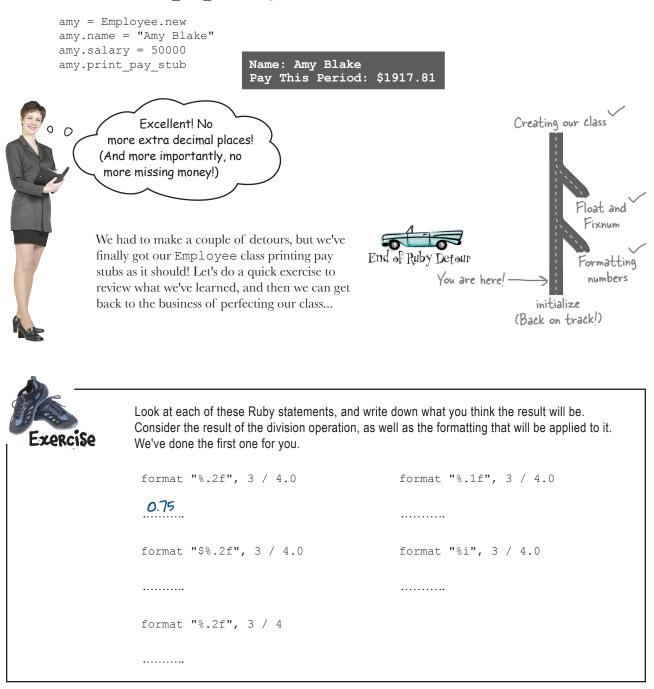
Let's try using format in the print pay stub method.

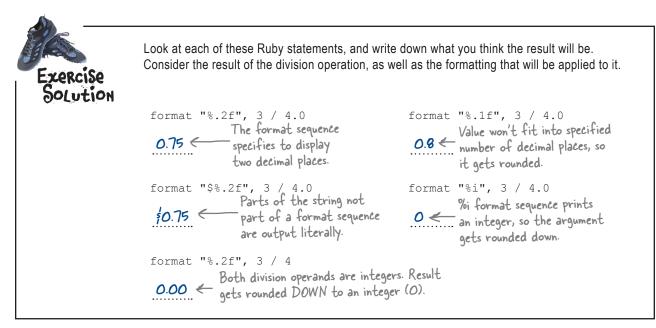
```
class Employee
...
def print_pay_stub
  puts "Name: #{@name}"
    pay_for_period = (@salary / 365.0) * 14
    formatted_pay = format("%.2f", pay_for_period)  Get a string with the pay amount
    puts "Pay This Period: $#{formatted_pay}"  rounded to 2 decimal places.
end
end
end
```



Using "format" to fix our pay stubs

We can test our revised print pay stub using the same values as before:





When we forget to set an object's attributes

Now that you have the employee pay printing in the correct format, you're puttering along, happily using your new Employee class to process payroll. Until, that is, you create a new Employee instance, and forget to set the name and salary attributes before calling print_pay_stub:

```
employee = Employee.new
employee.print_pay_stub
```



Woah! What happened? It's only natural that the name is empty; we forgot to set it. But what's this "undefined method for nil" error? What the heck is this nil thing?

This sort of error is pretty common in Ruby, so let's take a few pages to understand it.

"nil" stands for nothing

Now, let's create a new Employee instance, and call the revised method:

```
employee = Employee.new
employee.print_pay_stub
```



Well, that wasn't very helpful. Maybe we're missing something, though.

Back in Chapter 1, we learned that the inspect and p methods can reveal information that doesn't show up in ordinary output. Let's try again, using p:

We create another new instance, make another call to the instance method, and...

```
employee = Employee.new
employee.print_pay_stub
```



puts nil.to s

Ruby has a special value, nil, that represents *nothing*. That is, it represents the *absence* of a value.

Just because nil *represents* nothing doesn't mean it's *actually* nothing, though. Like everything else in Ruby, it's an object, and it has its own class:

puts nil.class NilClass

But if there's actually something there, how come we didn't see anything in the output?

It's because the to_s instance method from NilClass always returns an empty string.

Empty string!

The puts and print methods automatically call to_s on an object to convert it to a string for printing. That's why we got two blank lines when we tried to use puts to print the values of @name and @salary; both were set to nil, so we wound up printing two empty strings.

Unlike to_s, the inspect instance method from NilClass always returns the string "nil".

puts nil.inspect

nil

You may recall that the p method calls inspect on each object before printing it. That's why the nil values in @name and @salary appeared in the output once we called p on them.

"/" is a method

So, when you first create an instance of the Employee class, its @name and @salary instance variables have a value of nil. The @salary variable, in particular, causes problems if you call the print_pay_stub method without setting it first:

in `print pay stub': undefined method `/' for nil:NilClass Error: ----"nil" value! It's obvious from the error that the problem is related to the nil value. But it says undefined method '/'... Is division really a method? In Ruby, the answer is yes; most mathematical operators are implemented as methods. When Ruby sees something like this in your code: 6 + 2 The other operand is A method call! , passed as an argument. ... It converts it to a call to a method named + on the Fixnum object 6, with the object on the right of the + (that is, 2) as an argument: 6.+(2)Both forms are perfectly valid Ruby, and you can try running them yourself: puts 6 + 2 8 puts 6.+(2)The same is true for most of the Even comparison operators are other mathematical operators. implemented as methods. puts 7 - 3 4 puts 9 < 7 false puts 7.-(3)4 puts 9.<(7) false puts 3.0 * 2 puts 9 > 76.0 true puts 3.0.*(2) 6.0 puts 9.>(7) true puts 8.0 / 4.0 2.0 puts 8.0./(4.0) 2.0

But while the Fixnum and Float classes define these operator methods, NilClass does *not*.

puts nil./(365.0) Error: -----> undefined method `/' for nil:NilClass

In fact, nil doesn't define most of the instance methods you see on other Ruby objects.

And why should it? If you're doing mathematical operations with nil, it's almost certainly because you forgot to assign a value to one of the operands. You *want* an error to be raised, to bring your attention to the problem.

It was a mistake when we forgot to set a salary for an Employee, for example. And now that we understand the source of this error, it's time to prevent it from happening again.

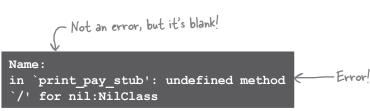
The "initialize" method

We tried to call print_pay_stub on an instance of our Employee class, but we got nil when we tried to access the @name and @salary instance variables.

```
employee = Employee.new
employee.print_pay_stub
```

Employee name salary print_pay_stub

Chaos ensued.



Here's the method where the nil values caused so much trouble:

def print_pay_stub puts "Name: #{@name}" pay_for_period = (@salary / 365.0) * 14 formatted_pay = format("\$%.2f", pay_for_period) puts "Pay This Period: #{formatted_pay}" end Results in call to "/" (actually an instance method) on @salary. Since it's nil, raises an error.

Here's the key problem... At the time we create an Employee instance, it's in an invalid state; it's not safe to call print_pay_stub until you set its @name and @salary instance variables.

If we could set @name and @salary *at the same time* as we create an Employee instance, it would reduce the potential for errors.

Ruby provides a mechanism to help with this situation: the initialize method. The initialize method is your chance to step in and make the object safe to use, before anyone else attempts to call methods on it.

```
class MyClass
  def initialize
    puts "Setting up new instance!"
    end
end
```

When you call MyClass.new, Ruby allocates some memory to hold a new MyClass object, then calls the initialize instance method on that new object.

MyClass.new Setting up new instance!

you are here > 123

Employee safety with "initialize"

Let's add an initialize method that will set up @name and @salary for new Employee instances before any other instance methods are called.

```
class Employee
              attr reader :name, :salary
              def name=(name)
                if name == ""
                  raise "Name can't be blank!"
                end
                @name = name
              end
              def salary=(salary)
                if salary < 0
                  raise "A salary of #{salary} isn't valid!"
                end
                @salary = salary
              end
              Qname = "Anonymous" Set the Qname instance variable.
Qsalary = 0.0
              [def initialize
Our new method.
              lend
                                  Set the Osalary instance variable.
              def print pay stub
                puts "Name: #{@name}"
                pay for period = (@salary / 365.0) * 14
                formatted pay = format("$%.2f", pay for period)
                puts "Pay This Period: #{formatted pay}"
              end
```

end

Now that we've set up an initialize method, @name and @salary will already be set for any new Employee instance. It'll be safe to call print pay stub on them immediately!

employee = Employee.new
employee.print_pay_stub

Name: Anonymous Pay This Period: \$0.00

Arguments to "initialize"

Our initialize method now sets a default @name of "Anonymous" and a default @salary of 0.0. It would be better if we could supply a value other than these defaults.

It's for situations like this that any arguments to the new method are passed on to initialize.

```
class MyClass
  def initialize(my_param)
    puts "Got a parameter from 'new': #{my_param}"
    end
end
MyClass.new("hello")
Got a parameter from 'new': hello
```

We can use this feature to let the caller of Employee.new specify what the initial name and salary should be. All we have to do is take name and salary parameters, and use them to set the @name and @salary instance variables.

```
class Employee
...
def initialize(name, salary)
    @name = name Use the "name" parameter to set the "@name" instance variable.
    @salary = salary
    Use the "salary" parameter to set the "@salary" instance variable.
    ...
end
end
```

And just like that, we can set @name and @salary via arguments to Employee.new!

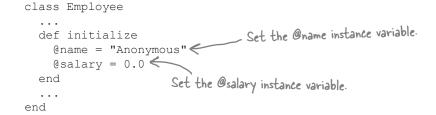
employee = Employee.new("Amy Blake", 50000) employee.print_pay_stub Forwarded to "initialize"! Name: Amy Blake Pay This Period: \$1917.81

Of course, once you set it up this way, you'll need to be careful. If you don't pass any arguments to new, there will be no arguments to forward on to initialize. At that point, you'll get the same result that happens any time you call a Ruby method with the wrong number of arguments: an error.

We'll look at a solution for this in a moment.

Using optional parameters with "initialize"

We started with an initialize method that set default values for our instance variables, but didn't let you specify your own...



Then we added parameters to initialize, which meant that you *had* to specify your own name and salary values, and couldn't rely on the defaults...

```
class Employee
...
def initialize(name, salary)
    @name = name  Use the "name" parameter to set the "@name" instance variable.
    @salary = salary
end
    Use the "salary" parameter to set the "@salary" instance variable.
end
```

Can we have the best of both worlds?

Yes! Since initialize is an ordinary method, it can utilize all the features of ordinary methods. And that includes optional parameters. (Remember those from Chapter 2?)

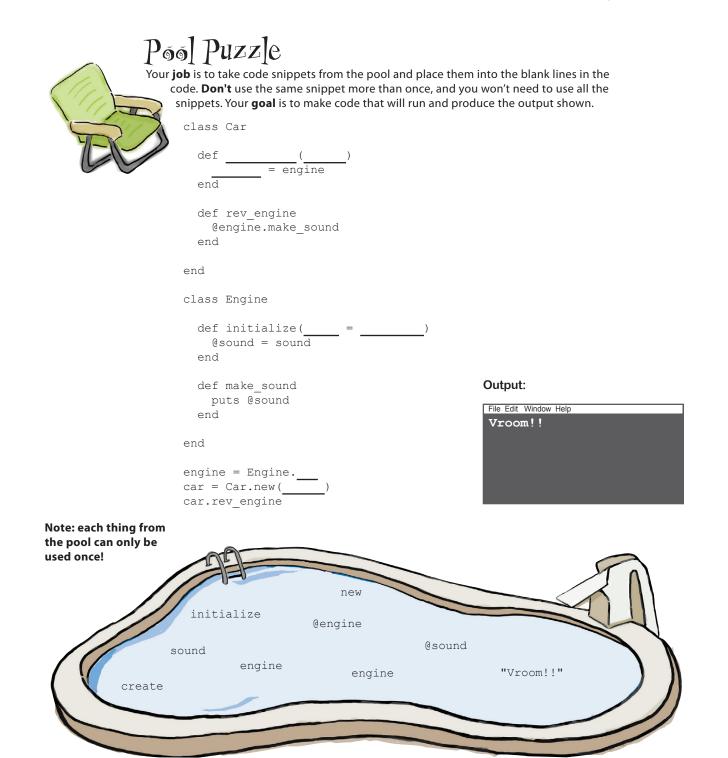
We can specify default values when declaring the parameters. When we omit an argument, we'll get the default value. Then, we just assign those parameters to the instance variables normally.

```
class Employee
...
def initialize(name = "Anonymous", salary = 0.0)
    @name = name
    @salary = salary
end
...
end
```

With this change in place, we can omit one or both arguments, and get the appropriate defaults!

```
Employee.new("Jane Doe", 50000).print_pay_stub
Employee.new("Jane Doe").print_pay_stub
Employee.new.print_pay_stub
```

Name: Jane Doe	
Pay This Period:	\$1917.81
Name: Jane Doe	
Pay This Period:	\$0.00
Name: Anonymous	
Pay This Period:	\$0.00



Poo] Puzzle Solution

class Car

def initialize (engine) @engine = engine end

def rev engine @engine.make sound end

end

engine = Engine.new car = Car.new(engine) car.rev engine

```
class Engine
  def initialize(sound = "Vroom!!")
    @sound = sound
  end
  def make sound
    puts @sound
  end
end
```

Output: File Edit Window Help

Vroom!!

Q: What's the difference between initialize methods in Ruby and constructors from other object-oriented languages?

A: They both serve the same basic purpose - to let the class prepare new instances for use. Whereas constructors are a special structure in most other languages, though, Ruby's initialize is just an ordinary instance method.

Q: Why do I have to call MyClass.new? Can't | just call initialize directly?

A: The new method is needed to actually create the object; initialize just sets up the new object's instance variables. Without new, there would be no object to initialize! For this reason, Ruby doesn't allow you to call the initialize method directly from outside an instance. (So, we oversimplified a little bit; initialize does differ from an ordinary instance method in one respect.)

bumb Questions

Q: Does MyClass.new always call initialize on the new object?

A: Yes, always.

Q: Then how have we been calling new on the classes we've made so far? They didn't have initialize methods!

A: Actually, they did have one... All Ruby classes inherit an initialize method from the Object superclass.

Q: But if Employee inherited an initialize method, why did we have to write our own?

A: The initialize from Object takes no arguments, and basically does nothing. It won't set up any instance variables for you; we had to override it with our own version in order to do that.

Q: Can I return a value from an initialize method?

A: You can, but Ruby will ignore it. The initialize method is intended solely for setting up new instances of your class, so if you need a return value, you should do that elsewhere in your code.

> The new method is needed to actually create the object; initialize just sets up the new object's instance variables.

"initialize" does an end-run around our validation



empty string as an Employee name:



There's also our salary= attribute writer method, which ensures that negative numbers aren't assigned as a salary:



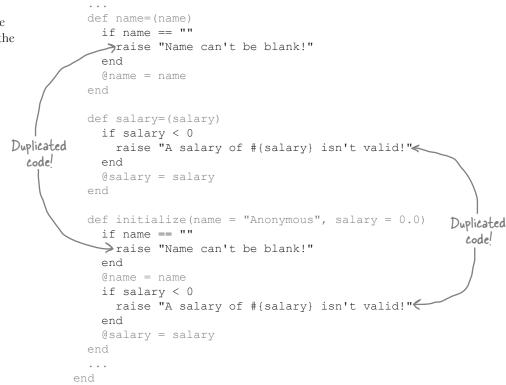
We have bad news for you... Since your initialize method assigns directly to the @name and @salary instance variables, bad data has a new way to sneak in!



"initialize" and validation

class Employee

We *could* get our initialize method to validate its parameters by adding the same validation code to the initialize method...



But duplicating code like that is a problem. What if we changed the initialize validation code later, but forgot to update the name= method? Rubyists try to follow the *DRY principle*, where DRY stands for Don't Repeat Yourself. It means that you should avoid duplicating code wherever possible, as it's likely to result in bugs.

What if we called the name= and salary= methods from *within* the initialize method? That would let us set the @name and @salary instance variables. It would also let us run the validation code, *without* duplicating it!

Calls between methods on the same instance with "self"

We need to call the name= and salary= attribute writer methods from within the initialize method of the same object. That will let us run the writer methods' validation code before we set the @name and @salary instance variables.

Unfortunately, code like this *won't* work...

```
class Employee
...
def initialize(name = "Anonymous", salary = 0.0)
    name = name  Doesn't work - Ruby thinks
    salary = salary you're assigning to a variable!
end
...
end
amy = Employee.new("Amy Blake", 50000)
amy.print_pay_stub
Name:
in `print_pay_stub': undefined method
`/' for nil:NilClass (NoMethodError)
```

The code in the initialize method treats name= and salary= *not* as calls to the attribute writer methods, but as re-setting the name and salary local variables to the same values they already contain! (If that sounds like a useless and nonsensical thing to do, that's because it is.)

What we *need* to do is make it clear to Ruby that we intend to call the name= and salary= instance methods. And to call an instance method, we usually use the dot operator.

```
But we're inside the initialize instance method... what would we put to the left of the dot operator?
```

We can't use the amy variable; it would be silly to refer to one instance of the class within the class itself. Besides, amy is out of scope within the initialize method.

```
class Employee
...
def initialize(name = "Anonymous", salary = 0.0)
Not in scope amy.name = name
amy.salary = salary
end
...
end
amy = Employee.new("Amy Blake", 50000)
Error: in `initialize': undefined local variable or method
```

`amy

Calls between methods on the same instance with "self" (cont.)

We need something to put to the left of the dot operator, so that we can call our Employee class's name= and salary= attribute accessor methods within our initialize method. The problem is, what do we put there? How do you refer to the current instance from *inside* an instance method?

```
class Employee
...
def initialize(name = "Anonymous", salary = 0.0)
Amy.name = name
amy.salary = salary
end
...
end
amy = Employee.new("Amy Blake", 50000)
```

Ruby has an answer: the self keyword. Within instance methods, self always refers to the current object.

We can demonstrate this with a simple class:

```
class MyClass
  def first_method
    puts "Current instance within first_method: #{self}"
    end
end
```

If we create an instance and call first_method on it, we'll see that inside the instance method, self refers to the object the method is being called on.

```
my_object = MyClass.new
puts "my_object refers to this object: #{my_object}"
my_object.first_method
```

my_object refers to this object: #<MyClass:0x007f91fb0ae508> _____ Current instance within first_method: #<MyClass:0x007f91fb0ae508> ____



The string representations of my_object and self include a unique identifier for the object. (We'll learn more about this much later, in the chapter on references.) The identifiers are the same, so it's the same object!

Same

Calls between methods on the same instance with "self" (cont.)

We can also use self with the dot operator to call a second instance method from inside the first one.

```
class MyClass
  def first method
    puts "Current instance within first method: #{self}"
    self.second method -
                         Calls here!
  end
  def second method 
    puts "Current instance within second method: #{self}"
  end
end
my object = MyClass.new
my object.first method
     Current instance within first method: #<MyClass:0x007ffd4b077510>
     Current instance within second method: #<MyClass:0x007ffd4b077510>
                                                                               -object!
```

Now that we have self to use the dot operator on, we can make it clear to Ruby that we want to call the name= and salary= instance methods, not to set the name and salary variables...

```
class Employee
  def initialize(name = "Anonymous", salary = 0.0)
    self.name = name - DEFINITELY a call to the "name=" method.
    self.salary = salary 🧲
  end
                             DEFINITELY a call to
  . . .
                             the "salary=" method.
end
```

Let's try calling our new constructor and see if it worked!

```
amy = Employee.new("Amy Blake", 50000)
amy.print pay stub
```

Name: Amy Blake Pay This Period: \$1917.81

Calls between methods on the same instance with "self" (cont.)

Success! Thanks to self and the dot operator, it's now clear to Ruby (and everyone else) that we're making calls to the attribute writer methods, not assigning to variables.

And since we're going through the accessor methods, that means the validation works, without any duplicated code!



When "self" is optional

Right now, our print_pay_stub method accesses the @name and @salary instance variables directly:

```
class Employee
def print_pay_stub
  puts "Name: #{@name}"
  pay_for_period = (@salary / 365.0) * 14
  formatted_pay = format("$%.2f", pay_for_period)
  puts "Pay This Period: #{formatted_pay}"
  end
```

end

But we defined name and salary attribute reader methods in our Employee class; we could use those instead of accessing the instance variables directly. (That way, if you ever change the name method to display last name first, or change the salary method to calculate salary according to an algorithm, the print_pay_stub code won't need to be updated.)

We *can* use the self keyword and the dot operator when calling name and salary, and it will work just fine:

```
class Employee
  attr_reader :name, :salary
  ...
  def print_pay_stub
    puts "Name: #{self.name}"
    pay_for_period = (self.salary / 365.0) * 14
    formatted_pay = format("$%.2f", pay_for_period)
    puts "Pay This Period: #{formatted_pay}"
  end
end
Employee.new("Amy Blake", 50000).print_pay_stub
Na
Pay
Pay
```

Name: Amy Blake Pay This Period: \$1917.81

When "self" is optional (cont.)

But Ruby has a rule that can save us a little typing when calling from one instance method to another... If you don't specify a receiver using the dot operator, the receiver defaults to the current object, self.

```
class Employee "self" omitted, still works!
...
def print_pay_stub ("self" omitted, still works!
puts "Name: #{name}" ("self" omitted, still works!
pay_for_period = (salary / 365.0) * 14
formatted_pay = format("$%.2f", pay_for_period)
puts "Pay This Period: #{formatted_pay}"
end
...
end
Employee.new("Amy Blake", 50000).print_pay_stub
Employee.new("Amy Blake", 50000).print_pay_stub
```

As we saw in the previous section, you *have* to include the self keyword when calling attribute writer methods, or Ruby will mistake the = for a variable assignment. But for any other kind of instance method call, you can leave self off, if you want.

Implementing hourly employees through inheritance

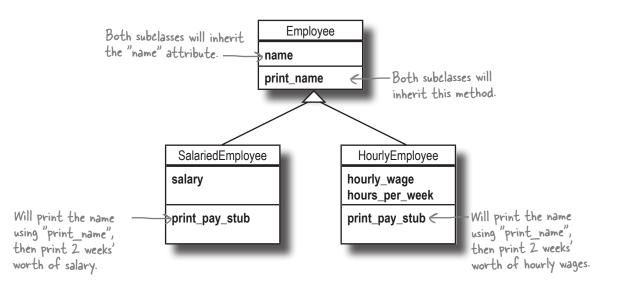
The Employee class you've created for Chargemore is working great! It prints accurate pay stubs that are formatted properly, and thanks to the initialize method you wrote, it's really easy to create new Employee instances.

But, at this point, it only handles salaried employees. It's time to look at adding support for employees that are paid by the hour.

The requirements for hourly employees are basically the same as for salaried ones; we need to be able to print pay stubs that include their name and the amount paid. The only difference is the way that we calculate their pay. For hourly employees, we multiply their hourly wage by the number of hours they work per week, then double that amount to get two weeks' worth.

Since salaried and hourly employees are so similar, it makes sense to put the shared functionality in a superclass. Then, we'll make two subclasses that hold the different pay calculation logic. (salary / 365.0) * 14 Salaried employee pay calculation formula

hourly_wage * hours_per_week * 2 Hourly employee pay calculation formula



Implementing hourly employees through inheritance (cont.)

Let's start by ensuring the common logic between SalariedEmployee and HourlyEmployee stays in the Employee superclass.

Since pay stubs for both salaried and hourly employees need to include their names, we'll leave the name attribute in the superclass, for the subclasses to share. We'll move the code that prints the name into the print_name method in the superclass.

```
We'll move the logic to calculate
pay for salaried employees to the
SalariedEmployee class, but
we'll call the inherited print_name
method to print the employee name.
```

class SalariedEmployee < Employee
class SalariedEmployee < Employee
class SalariedEmployee < Employee
class SalariedEmployee < Employee
attr_reader :salary
def salary=(salary)
 # Code to validate and set @salary
end
def print_pay_stub
 def print_pay_stub
 print_name </pre>
Calls print_name method
print_name
Calls print_name method
print_name
Calls print_name method
print_name
Calls print_name method
print_name
formatted_pay = format("\$%.2f", pay_for_period)
puts "Pay This Period: #{formatted_pay}"
end

end

With those changes in place, we can create a new SalariedEmployee instance, set its name and salary, and print a pay stub as before:

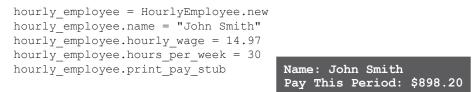
```
salaried_employee = SalariedEmployee.new
salaried_employee.name = "Jane Doe"
salaried_employee.salary = 50000
salaried_employee.print_pay_stub
Name: Jane Doe
Pay This Period: $1917.81
```

Implementing hourly employees through inheritance (cont.)

Now, we'll build a new HourlyEmployee class. It's just like SalariedEmployee, except that it holds an hourly wage and number of hours worked per week, and uses those to calculate pay for a two-week period. As with SalariedEmployee, storing and printing the employee name is left up to the Employee superclass.

```
class HourlyEmployee < Employee
attr_reader :hourly_wage, :hours_per_week
def hourly_wage=(hourly_wage)
    # Code to validate and set @hourly_wage
end
def hours_per_week=(hours_per_week)
    # Code to validate and set @hours_per_week
end
def print_pay_stub
    print_name
    pay_for_period = hourly_wage * hours_per_week * 2
    formatted_pay = format("$%.2f", pay_for_period)
    puts "Pay This Period: #{formatted_pay}"
end
end
```

And now we can create an HourlyEmployee instance. Instead of setting a salary, we set an hourly wage and number of hours per week. Those values are then used to calculate the pay stub amount.



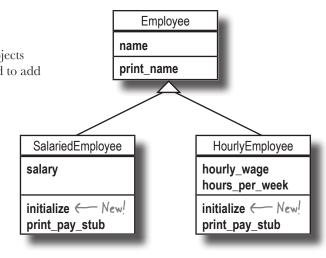
That wasn't bad at all! Through use of inheritance, we've implemented pay stubs for hourly employees, kept pay stubs for salaried employees, and minimized code duplication between the two.

We've lost something in the shuffle, though—our initialize method. We used to be able to set up an Employee object's data at the time we created it, and these new classes won't let us do that. We'll have to add initialize methods back in.

Restoring "initialize" methods

To make SalariedEmployee and HourlyEmployee objects that are safe to work with as soon as they're created, we'll need to add initialize methods to those two classes.

As we did with the Employee class before, our initialize methods will need to accept a parameter for each object attribute we want to set. The initialize method for SalariedEmployee will look just like it did for the old Employee class (since the attributes are the same), but initialize for HourlyEmployee will accept a different set of parameters (and set different attributes).



This is just like the initialize method for class SalariedEmployee < Employee the old Employee class. . . . def initialize(name = "Anonymous", salary = 0.0) < self.name = name self.salary = salary end . . . Again, we make end parameters optional by - providing defaults. class HourlyEmployee < Employee needs to self.name = name self.hourly wage = hourly wage accept 3 self.hours per week = hours per week parameters, end and set 3 . . . attributes. end

With our initialize methods added, we can once again pass arguments to the new method for each class. Our objects will be ready to use as soon as they're created. Salaried_employee = SalariedEmployee.new("Jane Doe", 50000) salaried_employee.print_pay_stub hourly_employee = HourlyEmployee.new("John Smith", 14.97, 30) hourly_employee.print_pay_stub

> Name: Jane Doe Pay This Period: \$1917.81 Name: John Smith Pay This Period: \$898.20

Inheritance and "initialize"

There's one small weakness in our new initialize methods, though: the code to set the employee name is duplicated between our two subclasses.

```
class SalariedEmployee < Employee
 def initialize(name = "Anonymous", salary = 0.0)
    self.name = name
    self.salary = salary
 end
  . . .
end
class HourlyEmployee < Employee
  . . .
 def initialize(name = "Anonymous", hourly_wage = 0.0, hours_per_week = 0.0)
    self.name = name - Duplicated in SalariedEmployee!*
    self.hourly wage = hourly wage
    self.hours per week = hours per week
 end
  . . .
end
```

In all other aspects of our subclasses, we delegate handling of the name attribute to the Employee superclass. We define the reader and writer methods there. We even print the name via the print_name method, which the subclasses call from their respective print_pay_stub methods.

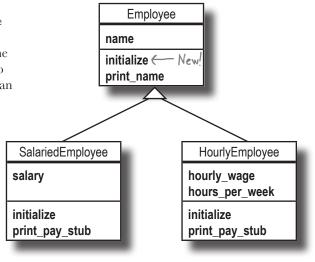
...But we don't do this for initialize. Could we?

Yes! We've said it before, and we'll say it again, initialize is just an *ordinary instance method*. That means that it gets inherited like any other, that means it can be overridden like any other, and it means that overriding methods can call it via super like any other. We'll demonstrate on the next page.

*Okay, we realize it's just one line of duplicated code. But the technique we're about to show you will work for much larger amounts of duplication.

"super" and "initialize"

To eliminate the repeated name setup code in our Employee subclasses, we can move the name handling to an initialize method in the superclass, then have the subclass initialize methods call it with super. SalariedEmployee will keep the logic to set up a salary, HourlyEmployee will keep the logic to set up an hourly wage and hours per week, and the two classes can delegate the shared logic for name to their shared superclass.



First, let's try moving the name handling from the initialize method in SalariedEmployee to the Employee class.

```
class Employee
  . . .
 def initialize (name = "Anonymous") - New initialize method that
                                        handles only the name!
   self.name = name
 end
  . . .
end
class SalariedEmployee < Employee
 . . .
 def initialize(name = "Anonymous", salary = 0.0)
   self.salary = salary Employee to set up the name.
 end
  . . .
end
```

Trying to use this revised initialize method reveals a problem, though...

```
salaried_employee = SalariedEmployee.new("Jane Doe", 50000)
salaried_employee.print_pay_stub
```

Error ----- in initialize: wrong number of arguments (2 for 0..1)

"super" and "initialize" (cont.)

Oops! We forgot a key detail about super that we learned earlier—if you don't specify a set of arguments, it calls the superclass method with the same set of arguments that the subclass method received. (This is true when using super in other instance methods, and it's true when using super within initialize.) The initialize method in SalariedEmployee received *two* parameters, and super passed them *both* on to the initialize method in Employee. (Even though it only accepts *one* argument.)

The fix, then, is to specify which parameter we want to pass on: the name parameter.

```
class SalariedEmployee < Employee
...
def initialize(name = "Anonymous", salary = 0.0)
    super(name) < Call "initialize" in Employee,
    self.salary = salary
    passing only the name.
...
end
...</pre>
```

Let's try to initialize a new SalariedEmployee again...

```
salaried_employee = SalariedEmployee.new("Jane Doe", 50000)
salaried_employee.print_pay_stub
Name: Jane Doe
Pay This Period: $1917.81
```

It worked! Let's make the same changes to the HourlyEmployee class...

```
class HourlyEmployee < Employee
...
def initialize(name = "Anonymous", hourly_wage = 0.0, hours_per_week = 0.0)
super(name)
self.hourly_wage = hourly_wage
self.hours_per_week = hours_per_week
end
...
end
hourly_employee = HourlyEmployee.new("John Smith", 14.97, 30)
hourly_employee.print_pay_stub
Name: John Smith
Pay This Period: $898.20
```

Previously, we used super within our print_pay_stub methods in SalariedEmployee and HourlyEmployee to delegate printing of the employee name to the Employee superclass. Now, we've just done the same thing with the initialize method, allowing the superclass to handle setting of the name attribute.

Why does it work? Because initialize is an instance method just like any other. Any feature of Ruby that you can use with an ordinary instance method, you can use with initialize.

there are no Dumb Questions

Q: If I override initialize in a subclass, does the superclass's initialize method run when the overriding initialize method runs?

A: Not unless you explicitly call it with the super keyword, no. Remember, in Ruby, initialize is just an ordinary method, like any other. If you call the move method on a Dog instance, does move from the Animal class get run as well? No, not unless you use super. It's no different with the initialize method.

Ruby is *not* the same as many other object-oriented languages, which automatically call the superclass's constructor before calling the subclass constructor.

Q: If I use super to call the superclass's initialize method explicitly, does it have to be the first thing I do in the subclass's initialize method?

A: If your subclass depends on instance variables that are set up by the superclass's initialize method, then you may want to invoke super before doing anything else. But Ruby doesn't require it. As with other methods, you can invoke super anywhere you want within initialize. You say the superclass's initialize method doesn't get run unless you call super... If that's true, then how does @last_name get set in this sample?

```
class Parent
  attr_accessor :last_name
  def initialize(last_name)
    @last_name = last_name
  end
end
class Child < Parent
end
child = Child now("Smith"))</pre>
```

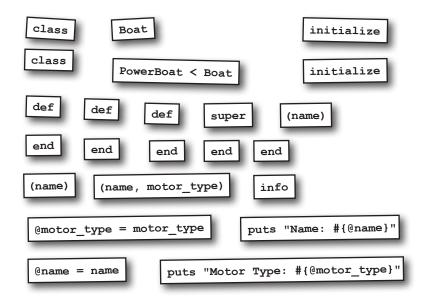
```
child = Child.new("Smith")
puts child.last_name
```

A: Because initialize is inherited from the Parent class. With Ruby instance methods, you only need to call super to invoke the parent class's method if you want it to run, and you've overridden it in the subclass. If you haven't overridden it, then the inherited method is run directly. This works the same for initialize as it does for any other method.



Code Magnets

A Ruby program is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working superclass and subclass, so the sample code below can execute and produce the given output?



Sample code:

boat = PowerBoat.new("Guppy", "outboard")
boat.info

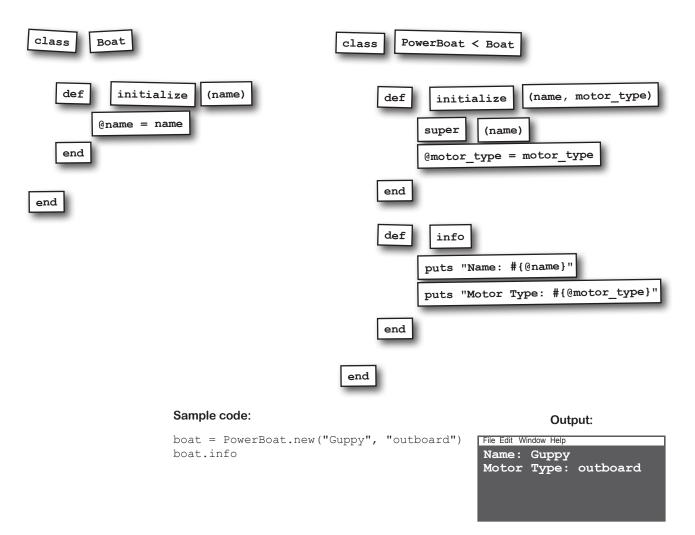
Output:

File Edit Window Help Name: Guppy Motor Type: outboard



Code Magnets Solution

A Ruby program is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working superclass and subclass, so the sample code below can execute and produce the given output?



Ruby lets us use as many

Same class, same attribute values

With your HourlyEmployee class complete, Chargemore is ready to begin a hiring blitz to staff its new stores. Here's the set of employees they need created for their first store downtown:

	Ç	space characters as we want, so we've aligned this - code for easier reading.
ivan	= HourlyEmployee.new("Ivan Stokes",	12.75, 25)
harold	<pre>= HourlyEmployee.new("Harold Nguyen",</pre>	12.75, 25)
tamara	<pre>= HourlyEmployee.new("Tamara Wells",</pre>	12.75, 25)
susie	= HourlyEmployee.new("Susie Powell",	12.75, 25)
edwin	= HourlyEmployee.new("Edwin Burgess",	10.50, 20)
ethel	= HourlyEmployee.new("Ethel Harris",	10.50, 20)
angela	= HourlyEmployee.new("Angela Matthews",	19.25, 30)
-	= HourlyEmployee.new("Stewart Sanchez",	

If you look at the above code, you'll probably notice there are large groups of objects where similar arguments are passed to the new method. There's a good reason for this: the first group are cashiers for the new store, the second group are janitors, and the third group are security guards.

Chargemore starts all new cashiers off at the same base pay and number of hours per week. Janitors get a different rate and number of hours than cashiers, but it's the same for all janitors. And the same is true for security guards. (Individuals may get raises later, depending on performance, but they all start out the same.)

The upshot is that there's a lot of repetition of arguments in those calls to new, and a lot of chances to make a typo. And this is just the first wave of hiring, for the first Chargemore store, so things can only get worse. Seems like we can make this easier.

An inefficient factory method

When we need to make many instances of a class that have similar data, you can often save some repetition by making a *factory method* to create objects pre-populated with the needed attribute values. (Factory methods are a programming pattern that can be used in any object-oriented language, not just Ruby.)

But using only the tools we have now, any factory method we make will be inefficient at best.

To demonstrate what we mean, let's try making a method to set up new HourlyEmployee objects with the default pay and hours per week for cashiers.

```
class HourlyEmployee
...
def turn_into_cashier
   self.hourly_wage = 12.75 Set hourly wage.
   self.hours_per_week = 25
end
...
end
ivan = HourlyEmployee.new("Ivan Stokes")
ivan.turn_into_cashier
ivan.print_pay_stub
Name: Ivan Stokes
Pay This Period: $637.50
```

This works, yes. So what's so inefficient about it? Let's look at our initialize method (which of course has to run when we create a new HourlyEmployee) again...

```
class HourlyEmployee
...
def initialize(name = "Anonymous", hourly_wage = 0, hours_per_week = 0)
    super(name)
    self.hourly_wage = hourly_wage Set hourly wage.
    self.hours_per_week = hours_per_week
end
...
end
```

We're setting the hourly_wage and hours_per_week attributes within initialize, then immediately turning around and setting them *again* within turn into cashier!

This is inefficient for Ruby, but there's potential for it to be inefficient for us, too. What if we didn't have default parameters for hourly_wage and hours_per_week on initialize? Then, *we'd* have to specify the arguments we're throwing away!

That's the problem with writing factory methods as instance methods: we're trying to *make* a new instance of the class, but there has to already *be* an instance to run the methods on! There must be a better way...

Fortunately, there is! Up next, we're going to learn about *class methods*.

Class methods

You don't *have* an instance of a class, but you *need* one. And you need a method to set it up for you. Where do you put that method?

You could stick it off by itself in some little Ruby source file, but it would be better to keep it together with the class that it makes instances of. You can't make it an instance method on that class, though. If you *had* an instance of the class, you wouldn't need to *make* one, now would you?

It's for situations like this that Ruby supports **class methods**—methods that you can invoke directly on a class, without the need for any instance of that class. You don't *have* to use a class method as a factory method, but they're *perfect* for the job.

A class method definition is very similar to any other method definition in Ruby. The difference: you specify that you're defining it *on the class itself*.

Within a class definition (but outside any instance method definitions), Ruby sets self to refer to the class that's being defined. So, many Rubyists prefer to replace the class name with self:

```
Also refers

to MyClass!

def self.my_class_method(p1, p2)

puts "Hello from MyClass!"

puts "My parameters: #{p1}, #{p2}"

end

end
```

In most ways, class method definitions behave just like you're used to:

- You can put as many Ruby statements as you like in the method body.
- You can return a value with the return keyword. If you don't, the value of the last expression in the method body is used as the return value.
- You can optionally define one or more parameters that the method accepts, and you can make the parameters optional by defining defaults.

Class methods (cont.)

We've defined a new class, MyClass, with a single class method:

```
def self.my_class_method(p1, p2)
  puts "Hello from MyClass!"
  puts "My parameters: #{p1}, #{p2}"
end
```

end

class MyClass

Once a class method is defined, you can call it directly on the class:

MyClass.my class method(1, 2)

Hello from MyClass! My parameters: 1, 2

Perhaps that syntax for calling a class method looks familiar to you...

MyClass.new

That's right, new is a class method! If you think about it, that makes sense; new can't be an *instance* method, because you're calling it to *get* an instance in the first place! Instead, you have to ask the *class* for a new instance of itself.

Now that we know how to create class methods, let's see if we can write some factory methods that will create new HourlyEmployee objects with the pay rate and hours per week already populated for us. We need methods to set up predefined pay and hours for three positions: cashier, janitor, and security guard.

```
class HourlyEmployee < Employee Accept the employee

name as a parameter.

def self.security_guard(name)

HourlyEmployee.new(<u>name</u>, <u>19.25</u>, <u>30</u>) Use predefined hourly_wage

end Use the given name to and hours_per_week for

def self.cashier(name)

HourlyEmployee.new(<u>name</u>, <u>12.75</u>, <u>25</u>) Same for the

end cashiers.

def self.janitor(name)

HourlyEmployee.new(<u>name</u>, <u>10.50</u>, <u>20</u>) Same for the

initors.

end
```

We won't know the name of the employee in advance, so we accept that as a parameter to each of the class methods. We *do* know the values for hourly_wage and hours_per_week for each employee position, though. We pass those three arguments to the new method for the class, and get a new HourlyEmployee object back. That new object is then returned from the class method.

Class methods (cont.)

Now, we can call the factory methods directly on the class, providing only the employee name.

```
angela = HourlyEmployee.security_guard("Angela Matthews")
edwin = HourlyEmployee.janitor("Edwin Burgess")
ivan = HourlyEmployee.cashier("Ivan Stokes")
```

The HourlyEmployee instances returned are fully configured with the name we provided, and the appropriate hourly_wage and hours_per_week for the position. We can begin printing pay stubs for them right away!

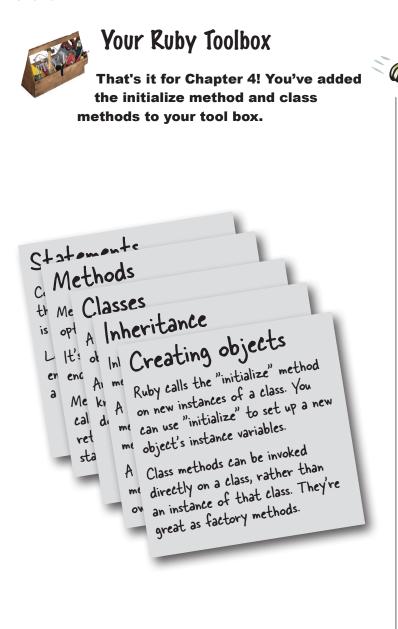
```
angela.print_pay_stub
edwin.print_pay_stub
ivan.print_pay_stub
Name: Angela Matthews
Pay This Period: $1155.00
Name: Edwin Burgess
Pay This Period: $420.00
Name: Ivan Stokes
Pay This Period: $637.50
```

In this chapter, you've learned that there are some pitfalls when creating new objects. But you've also learned techniques to ensure *your* objects are safe to use as soon as you make them. With well-designed initialize methods and factory methods, creating and configuring new objects is a snap!

Our complete source code

```
The "name" attribute is inherited by both
class Employee
                            - SalariedEmployee and HourlyEmployee.
 attr reader :name <
 def name=(name)
                                                                               employees.rb
   if name == ""
      raise "Name can't be blank!"
    end
                                               The "initialize" methods of both
   @name = name
                                               SalariedEmployee and HourlyEmployee
 end
                                             _ will call this method via "super".
 def initialize(name = "Anonymous") <</pre>
                         The "print_pay_stub" methods of both SalariedEmployee
   self.name = name
 end
                       - and Hourly Employee will call this method.
 def print name 🗲
   puts "Name: #{name}"
 end
end
class SalariedEmployee < Employee
                            — This attribute is specific to salaried employees.
 attr_reader :salary <
 def salary=(salary)
   if salary < 0
      raise "A salary of #{salary} isn't valid!"
    end
    @salary = salary
              - Called when we call "SalariedEmployee.new".
  end
 def initialize(name = "Anonymous", salary = 0.0)
    super (name) Call the superclass's "initialize" method, passing only the name.
    self.salary = salary - Set the salary ourselves, since it's specific to this class.
 end
 def print pay stub
   print_name - Have the superclass print the name. Calculate 2 weeks' pay.
   pay for period = (salary / 365.0) * 14
   puts "Pay This Period: #{formatted pay}"
                                                              2 decimal places.
 end
end
                                                                        Continued on next page!
```

```
class HourlyEmployee < Employee
                                      - Define a new class method
 def self.security guard(name) <
   HourlyEmployee.new (name, 19.25, 30) Create a new instance with the
                                              specified name, and a predefined
 end
                                                                            employees.rb
 def self.cashier(name)
                                              hourly wage and hours per week.
                                                                             (continued)
   HourlyEmployee.new(name, 12.75, 25)
 end
                                            .'Do the same as above for the
 def self.janitor(name) <----
                                              other hourly employee types.
   HourlyEmployee.new(name, 10.50, 20)
 attr_reader : hourly_wage, : hours_per_week < These attributes are specific to hourly employees.
 end
 def hourly wage=(hourly wage)
   if hourly wage < 0
     raise "An hourly wage of #{hourly wage} isn't valid!"
   end
   @hourly wage = hourly wage
 end
 def hours per week=(hours per week)
   if hours per week < 0
     raise "#{hours per week} hours per week isn't valid!"
   end
    @hours per week = hours per week
             Called when we call "HourlyEmployee.new".
 end
 def initialize(name = "Anonymous", hourly wage = 0.0, hours per week = 0.0)
   self.hourly wage = hourly wage <----</pre>
   self.hourly_wage = hourly_wage Set these ourselves, since
self.hours_per_week = hours_per_week they're specific to this class.
 end
 def print pay stub
                                                          ____ Calculate 2 weeks' pay.
   print name — Have the superclass print the name.
   pay_for_period = hourly_wage * hours per week * 2
   puts "Pay This Period: #{formatted pay}"
                                                           2 decimal places.
 end
end
jane = SalariedEmployee.new("Jane Doe", 50000)
jane.print pay stub
angela = HourlyEmployee.security guard("Angela Matthews")
ivan = HourlyEmployee.cashier("Ivan Stokes")
angela.print pay stub
ivan.print pay stub
```

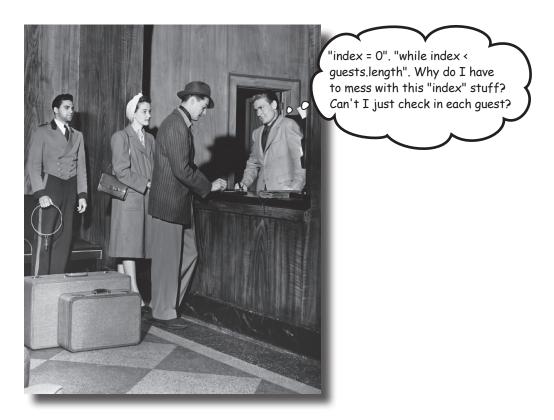


BULLET POINTS

- Number literals with a decimal point will be treated as Float instances. Without a decimal point, they'll be treated as Fixnum instances.
- If either operand in a mathematical operation is a Float, the result will be a Float.
- The format method uses format sequences to insert formatted values into a string.
- The format sequence type indicates the type of value that will be inserted. There are types for floating-point numbers, integers, strings, and more.
- The format sequence width determines the number characters a formatted value will take up within the string.
- The value nil represents nothing the absence of a value.
- Operators such as +, -, *, and / are implemented as methods in Ruby. When an operator is encountered in your code, it's converted into a method call.
- Within instance methods, the self keyword refers to the instance that the method is being called on.
- If you don't specify a receiver when calling an instance method, the receiver defaults to self.
- Within a class body, you can use either def ClassName.method_name or def self.method_name to define a class method.

5 arrays and blocks

It's Already Written



A whole lot of programming deals with lists of things. Lists of

addresses. Lists of phone numbers. Lists of products. Matz, the creator of Ruby, knew this. So he worked *really hard* to make sure that working with lists in Ruby is *really easy*. First, he ensured that **arrays**, which keep track of lists in Ruby, have lots of *powerful methods* to do almost anything you might need with a list.

Second, he realized that writing code to *loop over a list* to do something with each item, although tedious, is something developers were doing *a lot*. So he added **blocks** to the language, and removed the need for all that looping code. What is a block, exactly? Read on to find out...

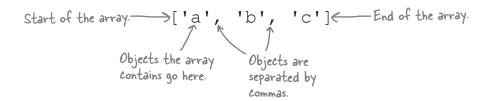
Arrays

Your new client is working on an invoicing program for an online store. They need three different methods, each of which works with the prices on an order. The first method needs to add all the prices together to calculate a total. The second will process a refund to the customer's account. And the third will take 1/3 off each price, and display the discount.



Hmm, so you have a list of prices (a *collection* of them, if you will), and you don't know in advance how many there will be... That means you can't use variables to store them - there's no way to know how many variables to create. You're going to need to store the prices in an *array*.

An **array** is used to hold a collection of objects. The collection can be any size you need.



Let's create an array to hold the prices from our first order now.

prices = [2.99, 25.00, 9.99]

You don't have to know an array's entire contents at the time you create it, though. You can also manipulate arrays *after* creating them...

Accessing arrays

So now we've got a place to store all our item prices. To retrieve the prices we stored in the array, we first have to specify which one we want.

Items in an array are numbered from left to right, starting with 0. This is called the array **index**.

[2.99, 25.00, 9.99] etc ... Index: To retrieve an item, you specify the integer index of the item you want within square brackets: prices [1] - Second item prices[2] Third item. So we can print out elements from our array like this. puts prices[0] 3.99 puts prices[2] 25.0 puts prices[1] 8.99 You can assign to a given array index with =, much like assigning to a variable. [0.99, 1.99, 2.99] If you assign to an index that's beyond the end - Here's the new element. of an array, the array will grow as necessary. prices[3] = 3.99[0.99, 1.99, 2.99, 3.99] p prices If you assign to an element that's *way* beyond the end of an array, it will still grow to accommodate your assignment. Here's the element "nil" means "there's There just won't be anything at the intervening indexes. we assigned to. nothing here"! prices[6] = 6.99[0.99, 1.99, 2.99, 3.99, nil, nil, 6.99] p prices Here, Ruby has placed nil (which, you may recall, represents the absence of a value) at the array indexes you haven't assigned to yet. You'll also get nil back if you access an element that's p prices[7] The array only extends through index 6! nil beyond the end of an array.

Arrays are objects, too!

Like everything else in Ruby, arrays are objects:

prices = [7.99, 25.00, 3.99, 9.99] puts prices.class Array

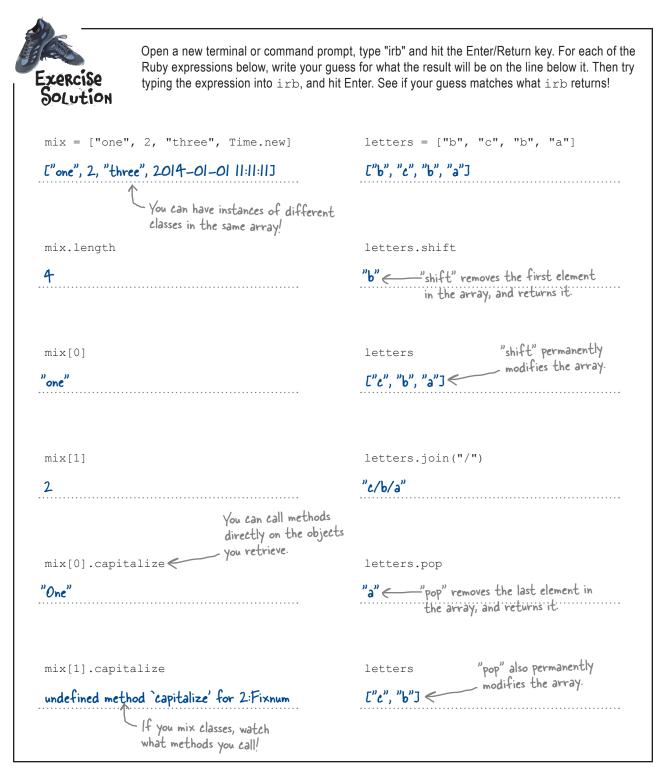
That means they have lots of useful methods attached directly to the array object. Here are some highlights...

Instead of using array indexes like prices[0], there are easy- to-read methods you can use:	<pre>puts prices.first 7.99 puts prices.last 9.99</pre>
There are methods to find out an array's size:	puts prices.length 4
There are methods to let you search for values within the array:	puts prices.include?(25.00) true
	<pre>puts prices.find_index(9.99) 3</pre>
There are methods that will let you insert or remove elements, causing t array to grow or shrink:	prices.push(0.99) the p prices [7.99, 25.0, 3.99, 9.99, 0.99]
, ,	prices.pop p prices [7.99, 25.0, 3.99, 9.99]
	prices.shift p prices [25.0, 3.99, 9.99]
The << operator (which, like most operators, is actually a method behi the scenes) also adds elements:	prices << 5.99 prices << 8.99 p prices [25.0, 3.99, 9.99, 5.99, 8.99]
Arrays have methods that can convert them to strings:	puts ["d", "o", "g"].join dog puts ["d", "o", "g"].join("-") d-o-g
And strings have methods that can convert them to arrays:	p "d-o-g".chars ["d", "-", "o", "-", "g"]
	p "d-o-g".split("-") ["d", "o", "g"]



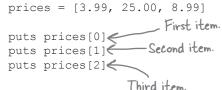
Open a new terminal or command prompt, type "irb" and hit the Enter/Return key. For each of the Ruby expressions below, write your guess for what the result will be on the line below it. Then try typing the expression into irb, and hit Enter. See if your guess matches what irb returns!

<pre>mix = ["one", 2, "three", Time.new]</pre>	letters = ["b", "c", "b", "a"]
mix.length	letters.shift
mix[0]	letters
mix[1]	letters.join("/")
<pre>mix[0].capitalize</pre>	letters.pop
mix[1].capitalize	letters



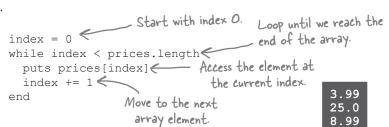
Looping over the items in an array

Right now, we can only access the particular array indexes that we specify in our code. Just to print all the prices in an array, we have to write this:



That won't work when the arrays get very large, or when we don't know their size beforehand.

But we can use a while loop to process *all* of an array's elements, one at a time.



	Calling the length instance me elements it holds, not the inde	thod on an array gets you the number of x of the last element.
Watch it!	So this code won't get you the last el	ement:
		p prices[prices.length] nil
	But this code will:	
	p prices[prices	.length - 1] 8.99
Likewise, a of the arra	a loop like this will go beyond the end y:	Because indexes start with zero, you need to ensure you're working with index numbers less than prices.length:
:	<pre>lex <= prices.length ices[index]</pre>	<pre>index = 0</pre>

The repeating loop

Now that we understand how to store the prices from an order in an array, and how to use a while loop to process each of those prices, it's time to work on the three methods your client needs:

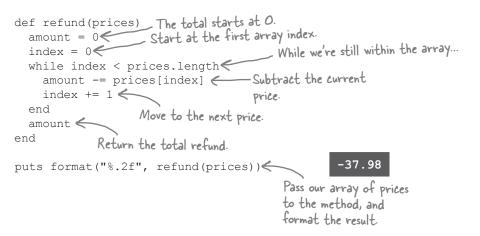
	一日 二日
 Given an array of prices, add them all together and return the total. Given an array of prices, subtract each price from the customer's account balance. Given an array of prices, reduce each item's price by 1/3, and print the savings. 	

The first requested feature is the ability to take these prices and total them. We'll create a method that keeps a running total of the amounts in an array. It will loop over each element in the array, and add it to a total (which we'll keep in a variable). After all the elements are processed, the method will return the total.

def total (prices) The total starts	s at O.
amount = 0 Start at the firs	st array index.
def total (prices) The total starts amount = 0 Start at the firs index = 0	While we're still within the array
while index < prices.length🧹	-
amount += prices[index] 🧲	-Add the current price
index += 1	to the total.
amount	
end Return the total.	Create an array holding prices from our order.
prices = [3.99, 25.00, 8.99]	
puts format("%.2f", total(prices	37.98
Ensure the correct number	Pass our array of prices to the method, and format the result.

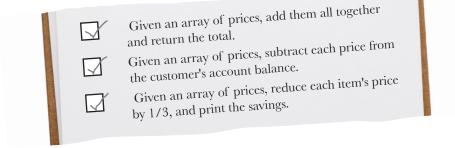
The repeating loop (cont.)

We need a second method that can process a refund for orders. It needs to loop through each item in an array, and subtract the amount from the customer's account balance.



Lastly, we need a third method that will reduce each item's price by 1/3 and print the savings.

That wasn't so bad! Looping over the items in the array let us implement all 3 of the methods your client needs!



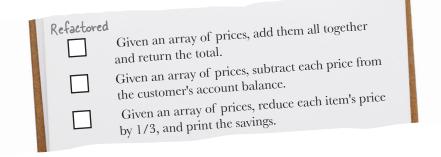
Your discount: \$3.00

The repeating loop (cont.)

If we look at the three methods together, though, you'll notice there's a *lot* of duplicated code. And it all seems to be related to looping through the array of prices. We've highlighted the duplicated lines below.

```
Highlighted lines are duplicated def total (prices)
amount = 0
index = 0
      while index < prices.length
This line in the middle ________amount += prices[index]
                               index += 1
         differs, though ...
                                  end
                                  amount
                               end
                               def refund(prices)
                                  amount = 0
                                  index = 0
                                  while index < prices.length
                   Differs... --- amount -= prices[index]
                                    index += 1
                                  end
                                  amount
                               end
                               def show discounts (prices)
                                  index = 0
                                  while index < prices.length
                        Differs... {amount_off = prices[index] / 3.0
puts format("Your discount: $%.2f", amount_off)
                                    index += 1
                                  end
                               end
```

This is definitely a violation of the DRY (Don't Repeat Yourself) principle. We need to go back to the drawing board and refactor these methods.



Eliminating repetition... the WRONG way...

Our total, refund, and show_discounts methods have a fair amount of repeated code related to looping over array elements. It would be nice if we could extract the repeated code out into another method, and have total, refund, and show discounts call it.

But a method that combines *all* the logic in total, refund, and show_variables wouldn't look very pretty... Sure, the code for the loop *itself* is repeated, but the code in the *middle* of the loop is all different. Also, the total and refund methods need a *variable* to track the total amount, but show_discounts doesn't.

Let's show you exactly *how* awful such a method would look. (We want you to fully appreciate it when we show you a better solution.) We'll try writing a method with an extra parameter, operation. We'll use the value in operation to switch which variables we use, and what code gets run in the middle of the loop.

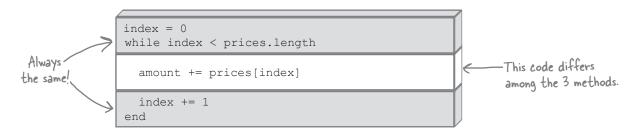
```
def do something with every item (array, operation) <
                                                                              "operation" should be set
                                                                              to "total", "refund", or
                        if operation == "total" or operation == "refund"
                       "show discounts". Don't
Here's the start of the
                                                                               make a typo!
loop - no more duplication! { index = 0
    while index < array.length</pre>
                         /if operation == "total"
                          amount += array[index]
                          elsif operation == "refund"
   Use the correct logic for amount -= array[index]
the current operation.
                            amount off = array[index] / 3.0
                            puts format("Your discount: $%.2f", amount off)
                         end
                          index += 1
                        end
                        if operation == "total" or operation == "refund"
                          return amount 🧲
                        end
                                            We don't return the value of this
                                            variable for "show discounts".
                      end
```

We warned you it would be bad. We've got if statements all over the place, each checking the value of the operation parameter. We've got an amount variable that we use in some cases, but not others. And we return a value in some cases, but not others. The code is ugly, and it's way too easy to make a mistake when calling it.

But if you *don't* write your code this way, how will you set up the variables you need prior to running the loop? And how will you execute the code you need in the *middle* of the loop?

Chunks of code?

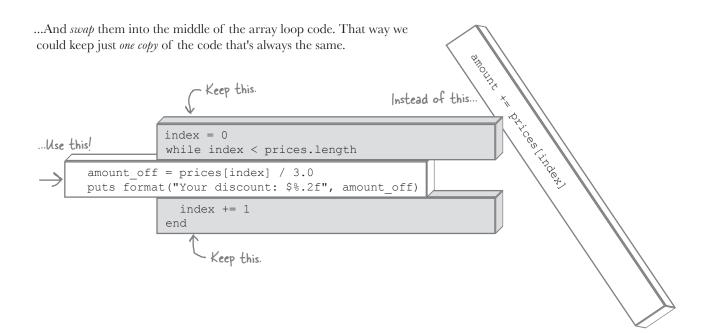
The problem is that the repeated code at the top and bottom of each method *surrounds* the code that needs to change.



It would sure be nice if we could take those other chunks of code that vary...

amount -= prices[index]

amount_off = prices[index] / 3.0
puts format("Your discount: \$%.2f", amount_off)



Blocks

What if we could pass a chunk of **code** into a method, like it was an argument? We could put the looping code at the top and bottom of the method, and then in the middle, we could run the code that was passed in!

It turns out we can do just that, using Ruby's blocks.

A **block** is a chunk of code that you associate with a method call. While the method runs, it can *invoke* (execute) the block one or more times. *Methods and blocks work in tandem to process your data.*

Blocks are mind-bending stuff. But stick with it!

We won't mince words. Blocks are going to be the hardest part of this book. Even if you've programmed in other languages, you've probably never seen anything like blocks. But *stick with it*, because the payoff is *big*.

Imagine if, for all the methods you have to write for the rest of your career, someone else *wrote half of the code for you*. For free. *They'd* write all the tedious stuff at the beginning and end, and just leave a little blank space in the middle for you to insert *your* code, the clever code, the code that runs your business.

If we told you that blocks can give you that, you'd be willing to do whatever it takes to learn them, right?

Well, here's what you'll have to do: be patient, and persistent. We're here to help. We'll look at each concept repeatedly, from different angles. We'll provide exercises for practice. Make sure to *do them*, because they'll help you understand and remember how blocks work.

A few hours of hard work now are going to pay dividends for the rest of your Ruby career, we promise. Let's get to it!



Defining a method that takes blocks

Blocks and methods work in tandem. In fact, you can't *have* a block without also having a method to accept it. So to start, let's define a method that works with blocks.

[On this page, we're going to show you how to use & to accept a block, and call to call it. This isn't the quickest way to work with blocks, but it DOES make it more obvious what's going on. We'll show you yield, which is more commonly used, in a few pages!]

Since we're just starting off, we'll keep it simple. The method will print a message, invoke the block it received, and print another message.

If you place an ampersand (&) before the last parameter in a method definition, Ruby will expect a block to be attached to any call to that method. It will take the block, convert it to an object, and store it in that parameter.

def my_method(&my_block)
...
end
When you call this method with a
block, it will be stored in "my_block".

Remember, a block is just a chunk of code that you pass into a method. To execute that code, stored blocks have a call instance method that you can call on them. The call method invokes the block's code.

```
def my_method(&my_block)

No ampersand; that's _____my_block.call <_____Run the block's code.

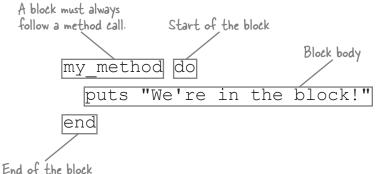
only used when defining ______

the parameter. end
```

OK, we know, you still haven't *seen* an actual block, and you're going crazy wondering what they look like. Now that the setup's out of the way, we can show you...

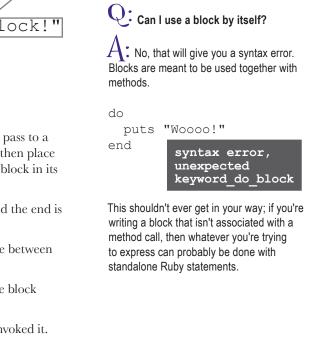
Your first block

Are you ready? Here it comes: your first glimpse of a Ruby block.



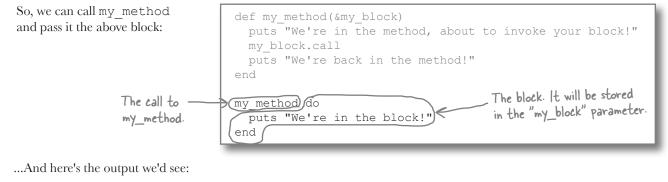
There it is! Like we said, a block is just a *chunk of code* that you pass to a method. We invoke my_method, which we just defined, and then place a block immediately following it. The method will receive the block in its my block parameter.

- The start of the block is marked with the keyword do, and the end is marked by the keyword end.
- The block *body* consists of one or more lines of Ruby code between do and end. You can place any code you like here.
- When the block is called from the method, the code in the block body will be executed.
- After the block runs, control returns to the method that invoked it.



there are no

Dumb Questions



We're in the method, about to invoke your block! We're in the block! We're back in the method!

Flow of control between a method and block

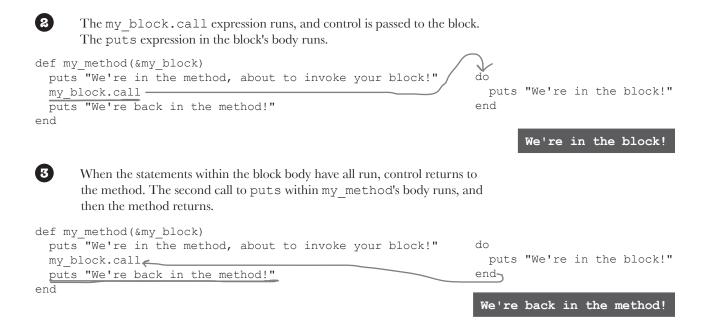
We declared a method named my_method, called it with a block, and got this output:

```
my_method do
puts "We're in the method, about to invoke your block!
end
We're in the block!
We're back in the method!
```

Let's break down what happened in the method and block, step by step.

1 The first puts statement in my_method's body runs.	
The method:	The block:
<pre>def my_method(&my_block) <u>puts "We're in the method, about to invoke your block!"</u> my_block.cal1 puts "We're back in the method!" ord</pre>	do puts "We're in the block!" end
end	

We're in the method, about to invoke your block!



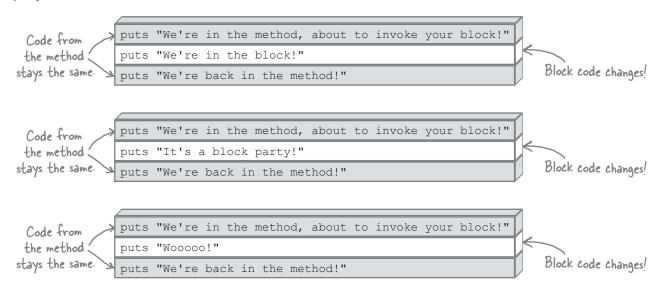
Calling the same method with different blocks

You can pass *many different blocks* to a *single method*.

We can pass different blocks to the method we just defined, and do different things:

	hod do "It's a block party!"
0	We're in the method, about to invoke your block! It's a block party! We're back in the method!
	hod do "Woocoo!"
	We're in the method, about to invoke your block! Wooooo! We're back in the method!

The code in the method is always the *same*, but you can *change* the code you provide in the block.



Calling a block multiple times

A method can invoke a block as many times as it wants.

This method is just like our previous one, except that it has *two* my block.call expressions:

```
      Declaring another
      def twice (&my_block)

      method that
      puts "In the method, about to call the block!"

      takes a block.
      puts "Back in the method, about to call the block again!"

      my_block.call
      Call the block AGAIN.

      puts "Back in the method, about to return!"
      end

      Calling the method
      twice do

      and passing it a block.
      puts "Woooo!"
```

The method name is appropriate: as you can see from the output, the method does indeed call our block twice!

In the method, about to call the block! Wooco! Back in the method, about to call the block again! Wooco! Back in the method, about to return!

```
1
      Statements in the method body run until the first my block.call
       expression is encountered. The block is then run. When it completes,
       control returns to the method.
def twice(&my block)
                                                                     do
  puts "In the method, about to call the block!"
                                                                       puts "Woooo!"
 my block.call Z
                                                                     end
  puts "Back in the method, about to call the block again!"
 my block.call
  puts "Back in the method, about to return!"
end
2
      The method body resumes running. When the second my block.call
```

The method body resumes running. When the second my_block.call expression is encountered, the block is run again. When it completes, control returns to the method so that any remaining statements there can run..

```
def twice(&my_block)
  puts "In the method, about to call the block!"
  my_block.call
  puts "Back in the method, about to call the block again!"
  puts "Back in the method, about to return!"
end
```

Block parameters

We learned back in Chapter 2 that when defining a Ruby method, you can specify that it will accept one or more parameters:

```
def print_parameters(p1, p2)
   puts p1, p2
end
```

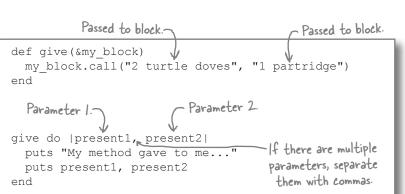
You're probably also aware that you can pass arguments when calling the method that will determine the value of those parameters.

```
print_parameters("one", "two")
```

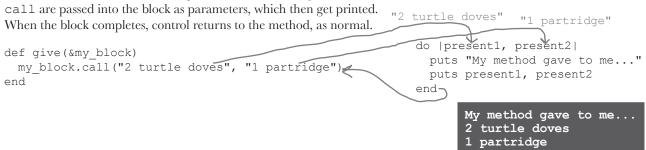
In a similar vein, a method can pass one or more arguments to a block. Block parameters are similar to method parameters; they're values that are passed in when the block is run, and that can be accessed within the block body.

Arguments to call get forwarded on to the block:

A block can accept one or more parameters from the method by defining them between vertical bar (|) characters at the start of the block:



So, when we call our method and provide a block, the arguments to



one

two

Q: Can I define a block once, and use it across many methods?

A: You can do something like this using Ruby procs (which are beyond the scope of this book). But it's not something you'll want to do in practice. A block is intimately tied to a particular method call, so much that a particular block will usually only work with a single method.

Q: Can a method take more than one block at the same time?

A: No. A single block is by far the most common use case, to the point that it's not worth the syntactic mess it would create for Ruby to support multiple blocks. If you ever want to do this, you could also use Ruby procs (but again, that's beyond the scope of this book).

Using the "yield" keyword

So far, we've been treating blocks like an argument to our methods. We've been declaring an extra method parameter that takes a block as an object, then using the call method on that object.

```
def twice(&my_block)
  my_block.call
  my_block.call
end
```

We mentioned that this wasn't the easiest way to accept blocks, though. Now, let's learn the less-obvious, but more-concise way: the yield keyword.

The yield keyword will find and invoke the block a method was called with—there's no need to declare a parameter to accept the block.

This method is functionally equivalent to the one above:

```
def twice
yield
yield
end
```

Just like with call, we can also give one or more arguments to yield, which will be passed to the block as parameters. Again, these methods are functionally equivalent:

```
def give(&my_block)
  my_block.call("2 turtle doves", "1 partridge")
end
def give
  yield "2 turtle doves", "1 partridge"
end
```



Declaring a &block parameter is useful in a few rare instances (which are beyond the scope of this book). But now that you understand what the yield keyword does, you should just use that in most cases. It's cleaner, and easier to read.

•

Block formats

So far, we've been using the do ... end def run block yield format for blocks. Ruby has a second block The do...end format end format, though: "curly-brace" style. You'll we've been using so far. see both formats being used "in the wild", so run block do 🗲 you should learn to recognize both. puts "do/end" end End of block. >run_block { puts "brackets" } "Curly-brace" format.-Block body, just like Aside from replacing do and end with curly brackets, with "do...end". the syntax and functionality are identical. do/end brackets And just as do ... end blocks can accept parameters, so can curly-brace blocks: def take this yield "present" end

```
take_this do |thing|
  puts "do/end block got #{thing}"
end
take this { |thing| puts "brackets block got #{thing}" }
```

By the way, you've probably noticed that all our do ... end blocks span multiple lines, but our curly-brace blocks all appear on a single line... This follows another convention that much of the Ruby community has adopted. It's valid *syntax* to do it the other way:

```
take_this { |thing|

puts "brackets: got #{thing}"

}

take_this do |thing| puts "do/end: got #{thing}" end

Breaks convention

(and is really ugly)! brackets: got present
```

But not only is that out of line with the convention, it's really ugly.

do/end block got present brackets block got present Conventional

Ruby blocks that fit on a single line should be surrounded with curly brackets. Blocks that span multiple lines should be surrounded with do ... end.

This is not the only convention for block formatting, but it is a common one.



 ${\tt Tonight's talk:}$ A method and a block talk about how they became associated with each other.

Method:

Hello, Block! I called you here tonight so we could educate people on how blocks and methods work together. I've had people ask me exactly what you contribute to the relationship, and I think we can clear those questions up for everyone.

So most parts of a method's job are pretty clearly defined. My task, for example, is to loop through each item in an array.

Sure! It's a task *lots* of developers need done; there's a lot of demand for my services. But then I encounter a problem: *what do I do* with each of those array elements? Every developer needs something different! And that's where blocks come in...

I know another method that does nothing but open and close a file. He's *very* good at that part of the task. But he has *no clue* what to do with the *contents* of the file...

I handle the general work that's needed on a *wide variety* of tasks...

Block:

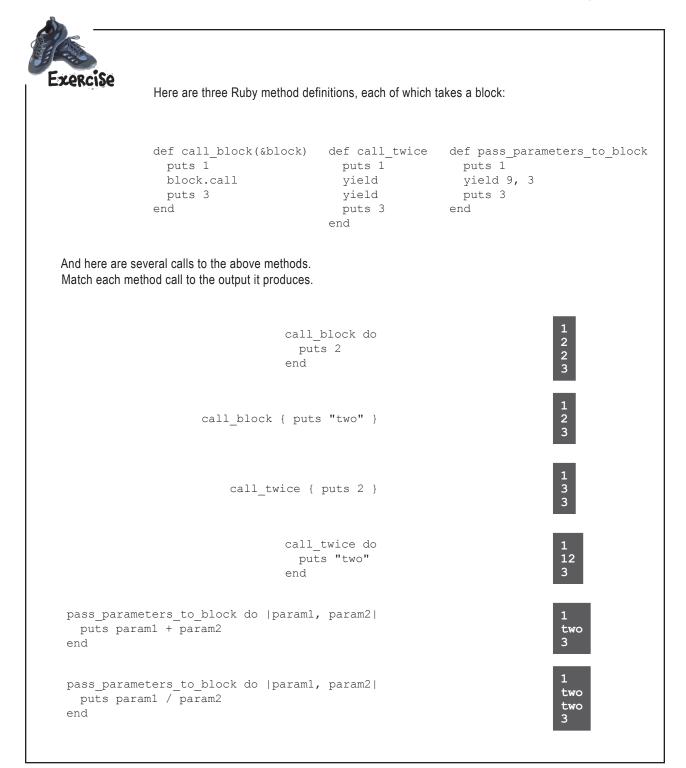
Sure, Method! I'm here to help whenever you call.

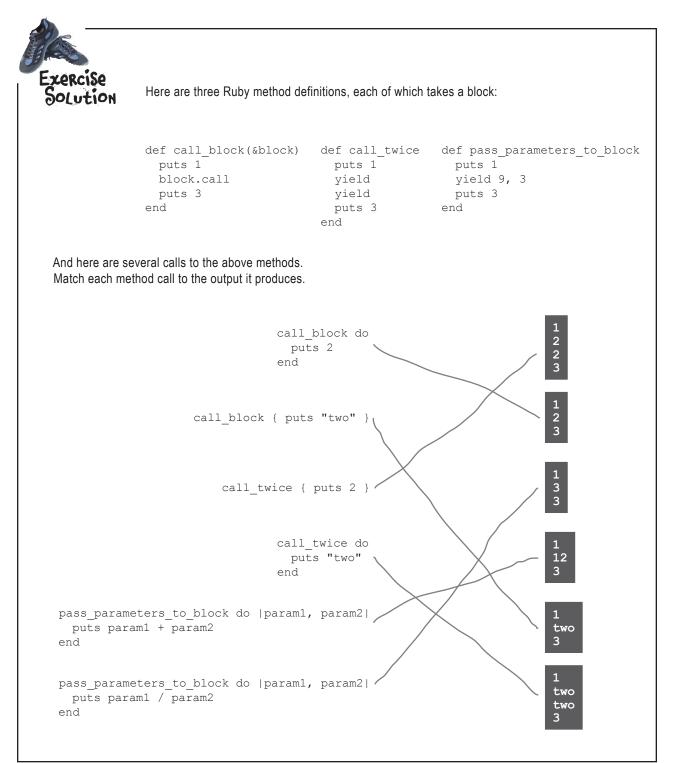
Right. Not a very glamorous job, but an important one.

Precisely. Every developer can write their *own* block that describes exactly what they need done with each element in the array.

...And so he calls on a block, right? And the block prints the file contents, or updates them, or whatever else the developer needs done. It's a great working relationship!

And I handle the logic that's specific to an *individual* task.





The "each" method

We had a lot to learn in order to get here: how to write a block, how a method calls a block, how a method can pass parameters to a block. And now, it's finally time to take a good, long look at the method that will let us get rid of that repeated loop code in our total, refund, and show discounts methods. It's an instance method that appears on every Array object, and it's called each.

You've seen that a method can yield to a block def my method more than once, with different values each time: yield 1 yield 2 yield 3 end my method { |param| puts param } 1 2 3 The each method uses this feature of Ruby to loop through each of the items in an array, vielding them to a block, one at a time. ["a", "b", "c"].each { |param| puts param }

class Array

If we were to write our own method that works like each, it would look very similar to the code we've been writing all along:

Remember, "self" refers to the current object. In this def each current element to a block! index += 1 🧲 end Then move to the next end element, just like before. end

We loop through each element in the array, just like in our total, refund, and show discounts methods. The key difference is that instead of putting code to process the current array element in the middle of the loop, we use the yield keyword to pass the element to a block.

а b С

The "each" method, step-by-step

We're using the each method and a block to process each of the items in an array:

["a", "b", "c"].each { |param| puts param }

Let's go step-by-step through each of the calls to the block, and see what it's doing.

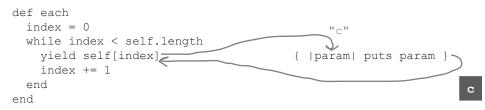
1

3

For the first pass through the while loop, index is set to 0, so the first element of the array gets yielded to the block as a parameter. In the block body, the parameter gets printed. Then control returns to the method, index gets incremented, and the while loop continues.

def each index = 0"a" while index < self.length vield self[index] { |param| puts param } index += 1end а end 2 Now, on the second pass through the while loop, index is set to 1, so the second element in the array will be yielded to the block as a parameter. As before, the block body prints the parameter, control then returns to the method, and the loop continues. def each index = 0"b" while index < self.length 1 yield self[index] { |param| puts param } index += 1end b end

After the third array element gets yielded to the block for printing and control returns to the method, the while loop ends, because we've reached the end of the array. No more loop iterations means no more calls to the block; we're done!

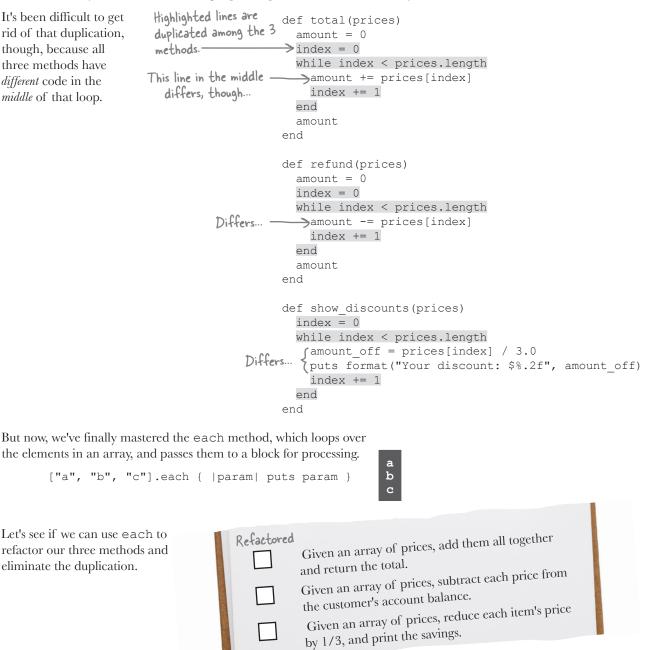


That's it! We've found a method that can handle the repeated looping code, and yet allows us to run our own code in the middle of the loop (using a block). Let's put it to use!

a b c

PRYing up our code with "each" and blocks

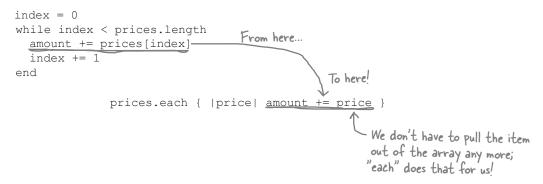
Our invoicing system requires us to implement these three methods. All three of them have nearly identical code for looping through the contents of an array.



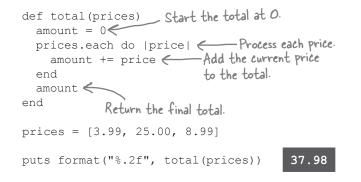
DRYing up our code with "each" and blocks (cont.)

First up for refactoring is the total method. Just like the others, it contains code for looping over prices stored in an array. In the middle of that looping code, total adds the current price to a total amount.

The each method looks like it will be perfect for getting rid of the repeated looping code! We can just take the code in the middle that adds to the total, and place in it a block that's passed to each.



Let's re-define our total method to utilize each, then try it out.



Perfect! There's our total amount. The each method worked!

DRYing up our code with "each" and blocks (cont.)

For each element in the array, each passes it as a parameter to the block. The code in the block adds the current array element to the amount variable, and then control returns back to each.

prices = [3.99, 25.00, 8.99]puts format("%.2f", total(prices)) 37.98 1 def each 3.99 index = 0while index < self.length do |price| yield self[index] amount += price index += 1endend end 2 def each 25.00 index = 0while index < self.length do |price| yield self[index] amount += price index += 1endend end 3 def each 8.99 index = 02 while index < self.length do |price| yield self[index] amount += price index += 1endend end We've successfully refactored Given an array of prices, add them all together Refactored the total method! \checkmark and return the total. Given an array of prices, subtract each price from But before we move on to the other two methods, the customer's account balance. let's take a closer look at Given an array of prices, reduce each item's price how that amount variable by 1/3, and print the savings. interacts with the block.

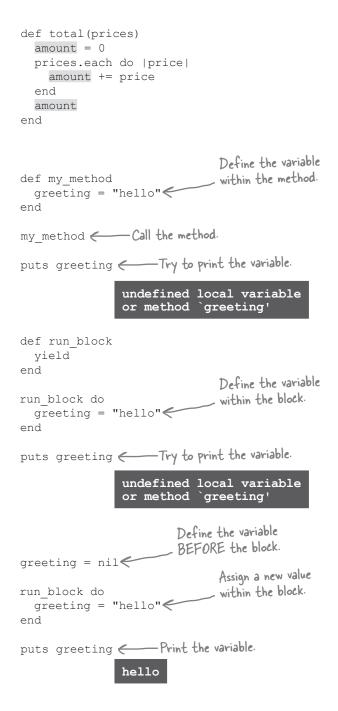
Blocks and variable scope

We should point something out about our new total method. Did you notice that we use the amount variable both *inside* and *outside* the block?

As you may remember from chapter 2, local variables defined within a method are *out of scope* as soon as the method ends. You can't access variables that are local to the method from *outside* the method.

The same is true of blocks, *if* you define the variable for the first time *inside* the block.

But, if you define a variable *before* a block, you can access it *inside* the block body. You can *also* continue to access it *after* the block ends!



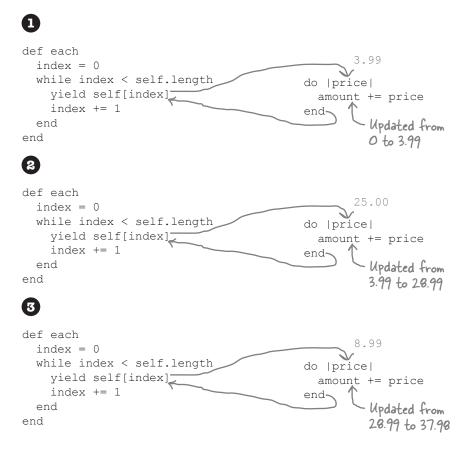
Blocks and variable scope (cont.)

Since Ruby blocks can access variables declared outside the block body, our total method is able to use each with a block to update the amount variable.

We can call total like this:

total([3.99, 25.00, 8.99])

The amount variable is set to 0, and then each is called on the array. Each of the values in the array are passed to the block. Each time the block is called, amount is updated:



When the each method completes, amount is still set to that final value, 37.98. It's that value that gets returned from the method.

```
def total(prices)
  amount = 0
  prices.each do |price|
    amount += price
  end
  amount
end
```

bumb Questions

Q: Why can blocks access variables that were declared outside their bodies, when methods can't? Isn't that unsafe?

A: A method can be accessed from other places in your program, far from where it was declared (maybe even in a different source file). A block, by contrast, is normally accessible only during the method call it's associated with. A block, and the variables it has access to, are all kept in the *same place* in your code. That means you can easily see all the variables a block is interacting with, meaning that accessing them is less prone to nasty surprises.

Using "each" with the "refund" method

We've revised the total method to get rid of the repeated loop code. We need to do the same with the refund and show_discounts methods, and then we'll be done!

The process of updating the refund method is very similar to the process we used for total. We simply take the specialized code from the middle of the generic loop code, and move it to a block that's passed to each.

def refund(prices)			
amount = 0		def refund(pri	ices)
index = 0	From To	amount = 0	
while index < prices.length	^h here her	prices.each	do price
amount -= prices[index]—		<u>amount -=</u>	price 🤶
index += 1		end	Again, we don't have
end		amount	
amount		end	to pull the item out
end			of the array; "each"
			gets it for us!

-37.98

Much cleaner, and calls to the method still work just the same as before!.

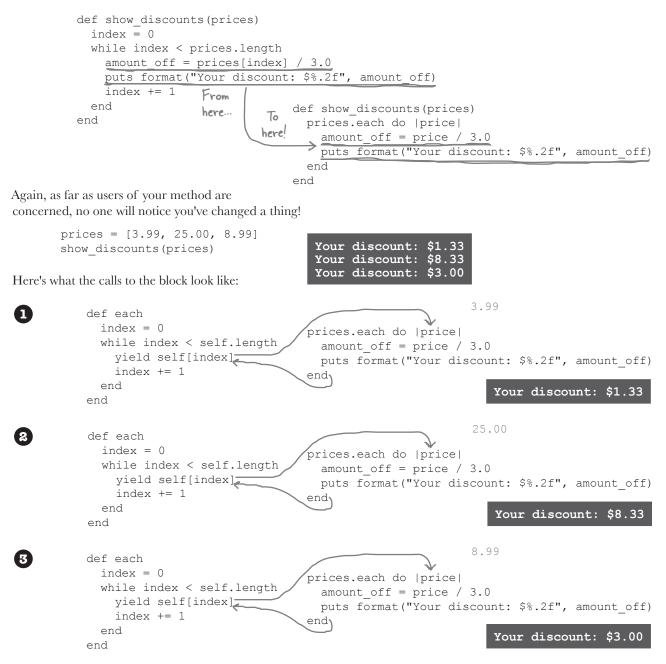
```
prices = [3.99, 25.00, 8.99]
puts format("%.2f", refund(prices))
```

Within the call to each and the block, the flow of control looks very similar to what we saw in the total method:

```
(\mathbf{1})
           def each
                                                               3.99
             index = 0
             while index < self.length
                                                       do |price|
               yield self[index]
                                                          amount -= price
               index += 1
                                                       end
                                                                            Updated from
             end
                                                                             0 to -3.99
           end
(2)
           def each
                                                               25.00
             index = 0
             while index < self.length
                                                       do |price|
               yield self[index]
                                                          amount -= price
               index += 1
                                                       end
                                                                              Updated from
             end
                                                                              -3.99 to -28.99
           end
3
           def each
                                                               8.99
             index = 0
             while index < self.length
                                                       do [price]
               yield self[index]
                                                          amount -= price
               index += 1
                                                        end
                                                                               Updated from
             end
                                                                              28.99 to -37.98
           end
```

Using "each" with our last method

One more method, and we're done! Again, with show_discounts, it's a matter of taking the code out of the middle of the loop, and moving it into a block that's passed to each.



Our complete invoicing methods

```
def total (prices) _ Start the total at O.
  amount = 0
  prices.each do |price| - Process each price.
    amount += price ---- Add the current price
  end
                           to the total.
  amount ←
end
             Return the final total
def refund (prices) _ Start the total at O.
  amount = 0
  prices.each do |price| - Process each price.
    amount -= price ---- Refund the current price.
  end
  amount 🤶
end
            Return the final total
def show discounts (prices)
  prices.each do |price| - Process each price.
    amount off = price / 3.0 Calculate discount.
    puts format("Your discount: $%.2f", amount off)
  end
               Format and print the current discount.
end
prices = [3.99, 25.00, 8.99]
puts format("%.2f", total(prices))
puts format("%.2f", refund(prices))
show discounts (prices)
```

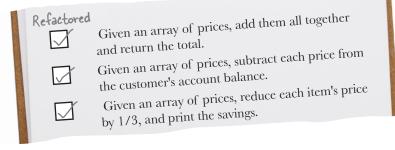


Save this code in a file named prices.rb. Then try running it from the command line!

37.98		rb
-37.9	98	
Your	discount:	\$1.33
Your	discount:	\$8.33
	discount:	

We've gotten rid of the repetitive loop code!

We've done it! We've refactored the repetitive loop code out of our methods! We were able to move the portion of the code that *differed* into blocks, and rely on a method, each, to replace the code that remained the *same!*



Pool Puzzle



Your **job** is to take code snippets from the pool and place them into the blank lines in the code. **Don't** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make code that will run and produce the output shown.

```
def pig latin (words)
         original length = 0
         = 0
        words. do
    puts "Original word: #{word}"
                 += word.length
           letters = word.chars
           first letter = letters.shift
           new word = "#{letters.join}#{first letter}ay"
           puts "Pig Latin word: #{ }"
                                                           Output:
                 += new word.length
         end
                                                           File Edit Window Help
                                                            Original word: blocks
         puts "Total original length: #{
                                                            Pig Latin word: locksbay
         puts "Total Pig Latin length: #{new length}"
                                                            Original word: totally
                                                            Pig Latin word: otallytay
                                                            Original word: rock
       end
                                                            Pig Latin word: ockray
                                                            Original total length: 17
       my words = ["blocks", "totally", "rock"]
                                                            Total Pig Latin length: 23
       pig_latin( )
Note: each thing from
the pool can only be
used once!
                                |word|
                                           yield
                    original length
                                        new length
                                                                   new word
                                                       shrink
                      new length
                                          my words
                                                                            new
             each
                                                         original length
```

Pool Puzzle Solution

def pig_latin(words)

```
original_length = 0
new_length = 0
```

```
words.each do |word|
puts "Original word: #{word}"
original_length += word.length
letters = word.chars
first_letter = letters.shift
new_word = "#{letters.join}#{first_letter}ay"
puts "Pig Latin word: #{new_word}"
new_length += new_word.length
end
```

puts "Total original length: #{original_length}"
puts "Total Pig Latin length: #{new length}"

end

my_words = ["blocks", "totally", "rock"]
pig_latin(my_words)

Output:

File Edit Window Help
Original word: blocks
Pig Latin word: locksbay
Original word: totally
Pig Latin word: otallytay
Original word: rock
Pig Latin word: ockray
Original total length: 17
Total Pig Latin length: 23

Utilities and Appliances, Blocks and Methods

Imagine two very different electric appliances: a mixer, and a drill. They have pretty different jobs: one is used for baking, the other for carpentry. And yet, they have a very similar need: electricity.

Now, imagine a world where, any time you wanted to use an electric mixer or drill, you had to wire your appliance into the power grid yourself. Sounds tedious (and fairly dangerous), right?

That's why, when your house was built, an electrician came and installed *power outlets* in every room. They provide the same utility (electricity) through the same interface (an electric plug) to very different appliances.

The electrician doesn't know the details of how your mixer or drill works, and he doesn't care. He just uses his skills and training to get the current safely from the electric grid to the outlet.

The designers of your appliance, likewise, have no idea how to wire a home for electricity. They just know how to take power from an outlet and use it to make their devices operate.

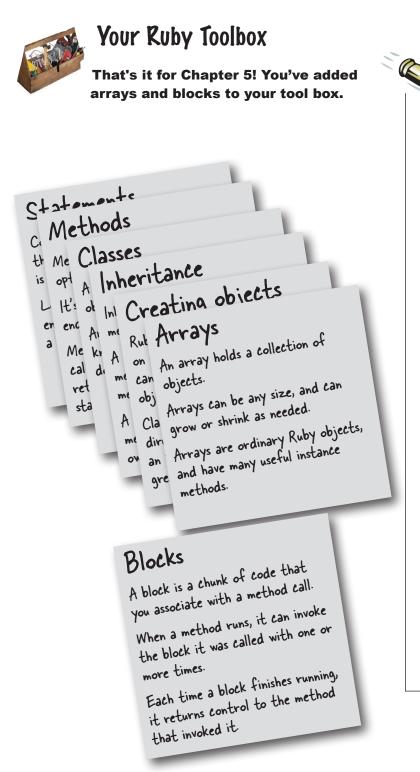
You can think of the author of a method that takes a block as being kind of like an electrician.	def wire
They don't know how the block works, and they don't care. They just use their knowledge of a	yield "current"
problem (say, looping through an array's elements) to get the neccessary data to the block.	end

You can think of calling a method with a block as being kind of like plugging an appliance into an outlet. Like the outlet supplying power, the block parameters offer a safe, consistent interface for the method to supply data to your block. Your block doesn't have to worry about how the data got there, it just has to process the parameters it's been handed.

	CL	ike a po	ower outle	et.			
	power	puts	"Using	#{power} #{power}			}
				g current g current			

Not every appliance uses electricity, of course; some require other utilities. There are stoves and furnaces that require gas. There are automatic sprinklers and spray nozzles that use water.

Just as there are many kinds of utilities to supply many kinds of appliances, there are many methods in Ruby that supply data to blocks. The each method was just the beginning. We'll be looking at some of the others over the next chapter.

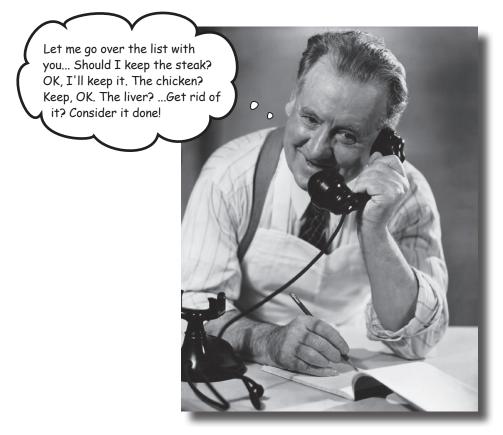


BULLET POINTS

- The index is a number that can be used to retrieve a particular item from an array. An array's index starts with 0.
- You can also use the index to assign a new value to a particular array location.
- The length method can be used to get the number of items in an array.
- Ruby blocks are only allowed following a method call.
- There are two ways to write a block: with do ... end or with curly brackets ({})
- You can specify that the last method parameter should be a block by preceding the parameter name with an ampersand (*&*).
- It's more common to use the yield keyword, though. You don't have to specify a method parameter to take the block - yield will find and invoke it for you.
- A block can receive one or more parameters from the method. Block parameters are similar to method parameters.
- A block can get or update the value of local variables that appear in the same scope as the block.
- Arrays have an each method which invokes a block once for each item in an array.

6 block return values

How Should I Handle This?



You've only seen a fraction of the power of blocks. Up until now,

the *methods* have just been handing data off to a *block.*, and expecting the block to do all the work with it. But a *block* can also return data back to the *method*. This feature lets the method get *directions* from the block, allowing it to do more of the work. In this chapter, we'll show you some methods that will let you take a *big*, *complicated* collection, and use **block return values** to cut it down to size. page goal header

A big collection of words to search through

Word got out on the great work you did on the invoicing program, and your next client has already come in - a movie studio. They release a lot of films each year, and the task of making commercials for all of them is enormous. They want you to write a program that will go through the text of movie reviews, find adjectives that describe a given movie, and generate a collage of those adjectives:

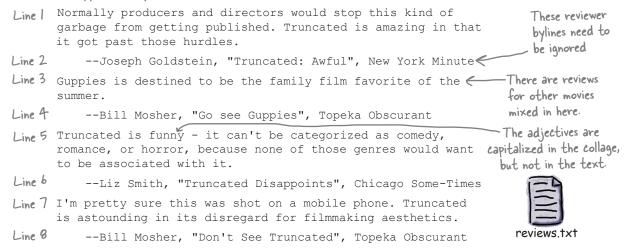
The critics agree, <u>Hindenburg</u> is: "Romantic" "Thrilling" "Explosive"



They've given you a sample text file to work off of, and they want you to see if you can make a collage for their new release, Truncated.

Looking at the file, though, you can see your work is cut out for you:

Lines are wrapped so they fit here ...



It's true, this job is a bit complex. But don't worry, arrays and blocks can help!

A big collection of words to search through (cont.)

Let's break our tasks down into a checklist:



Five tasks to accomplish. Sounds simple enough. Let's get to it!

Opening the file

Our first task is to open the text file with the review contents. This is easier than it sounds - Ruby has a built-in class named File that represents files on disk. To open a file named "reviews.txt" in the current directory (folder) so you can read data from it, call the open method on the File class:

review_file = File.open("reviews.txt")

The open method returns a new File object. (It actually calls File.new for you, and returns the result of that.)

puts review_file.class

File

There are many different methods that you can call on this File instance, but the most useful one for our current purpose is the readlines method, which returns all the lines in the file as an array.

```
lines = review_file.readlines
puts "Line 4: #{lines[3]}"
puts "Line 1: #{lines[0]}"
```

```
Line 4: --Bill Mosher, "Go see Guppies",
Topeka Obscurant
Line 1: Normally producers and directors would
stop this kind of garbage from getting published.
Truncated is amazing in that it got past those
hurdles.
```

- (Wrapped to fit this page.)

Safely closing the file

We've opened the file, and read its contents. Your next step should be to *close the file*. Closing the file tells the operating system, "I'm done with this file; others can use it now."

review_file.close

Why are we so emphatic about doing this? Because *bad things happen* when you forget to close files.

You can get errors if your operating system detects that you have too many files open at once. If you try to read all the contents of the same file multiple times without closing it, it will appear to be empty on subsequent attempts (because you've already read to the end of the file, and there's nothing after that). If you're writing to a file, no other program can see the changes you made until you *close the file*. It is *very important* not to forget.

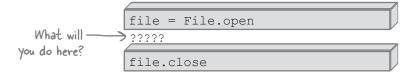
Are we making you nervous? Don't be. As usual, Ruby has a developer-friendly solution to this problem.

Safely closing the file, with a block

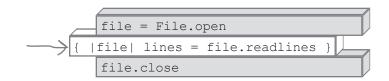
Ruby offers a way to open a file, do whatever you need with it, and *automatically* close it again when you're done with it. The secret is to call File.open... with *a block*!

```
File object is returned
We just change our
                                  and needs to be stored
code from this:
                                 - in a variable.
                        review file = File.open("reviews.txt")
                        lines = review file.readlines
                        review file.close
                                             Need to call "close"
                                             when done.
                                                       File object is passed
... To this!
                                                       as a parameter to
                                                       the block.
                        File.open("reviews.txt") do |review file|
                          lines = review file.readlines
                        end
                             When the block finishes,
                             the file is automatically
                             closed for you!
```

Why does File.open use a block for this purpose? Well, the first and last steps in the process are pretty well-defined:



...But the creators of File.open have *no idea* what you intend to do with that file while it's open. Will you read it one line at a time? All at once? That's why they let *you* decide what to do, by passing in a block.



Pon't forget about variable scope!

When we're *not* using a block, we can access the array of lines from the File object just fine.

```
review file = File.open("reviews.txt")
lines = review file.readlines
review file.close
                      8
```

puts lines.length

Switching to the block form of File.open has introduced a problem, however. We store the array returned by readlines in a variable *within* the block, but we can't access it *after* the block.

```
File.open("reviews.txt") do |review file|
 lines = review file.readlines
end
puts lines.length
                   undefined local variable
                   or method `lines'
```

The problem is that we're *creating* the lines variable *within* the block. As we learned back in Chapter 5, any variable created within a block has a scope that's limited to within the block. Those variables can't be "seen" from outside the block.

But, as we also learned in Chapter 5, local variables declared before a block can be seen within the block body (and are still visible after the block, of course). So the simplest solution is to create the lines variable *before* declaring the block.

```
lines = []
File.open("reviews.txt") do |review file|
  lines = review_file.readlines - Still in scope.
end
puts lines.length
        Still in scope!
```

OK, we've safely closed the file, and we've got our review contents. What do we do with them? We'll be tackling that problem next.

there are no Dumb Questions

Q: How can File.open work both with a block and without one?

 ${
m A}$: Within a Ruby method, you can call the block given? method to check whether the method caller used a block, and change the method behavior accordingly.

If we were coding our own (simplified) version of File.open, it might look like this:

```
def File.open(name, mode)
  file = File.new(name, mode)
  if block given?
    yield(file)
  else
    return file
  end
end
```

If a block is given, the file is passed to it for use within the block. If it's not, the file is returned.

Exa	Three Ruby scripts are below. Fill in th produce the specified output.	e blank in each script so that it will run successfully and
0	def yield_number yield 4 end	
	<pre>yield_number { number array << numbe p array [1, 2, 3, 4]</pre>	r }
		[1, 2, 3].each { number sum += number }
3	<pre>File.open("sample.txt") do file contents = file.readlines</pre>	puts sum 6
	end puts contents This is the first This is the secor This is the last	

Exe	Three Ruby scripts are below. Fill in produce the specified output.	n the blank in each script so that it will run successfully and
1	def yield_number yield 4 end	
	array = [1, 2, 3]	
	<pre>yield_number { number array << nur</pre>	aber }
	p array [1, 2, 3, 4]	sum = 0
	Any value at all wil	[1, 2, 3].each { number sum += number }
3	work here, since we contents = [] assign a completely new value in the bl File.open ("sample.txt") do file	puts sum 6
	<pre>contents = file.readlines end</pre>	
	puts contents This is the first This is the set This is the last	

Finding array elements we want, with a block

We've opened the file, and used the readlines method to get an array with every line from the file in its own element. The first feature from our checklist is complete!

Let's see what remains:	M	Get the file contents.
		Find reviews for the current movie.
		Discard reviewer bylines.
		Find an adjective within each review.
		Capitalize each adjective and put it in quotation
		marks.

It seems we can't expect the text file to contain only reviews for the movie we want. Reviews for other movies are mixed in there, too:

Line | Normally producers and directors would stop this kind of garbage from getting published. Truncated is amazing in that it got past those hurdles. Line 2 --Joseph Goldstein, "Truncated: Awful", New York Minute Line 3 Guppies is destined to be the family film favorite of the --A review for a completely summer. different moviel Line 4 --Bill Mosher, "Go see Guppies", Topeka Obscurant Line 5 Truncated is funny - it can't be categorized as comedy, romance, or horror, because none of those genres would want to be associated with it. Line 6 --Liz Smith, "Truncated Disappoints", Chicago Some-Times reviews.txt . . .

Fortunately, it also looks like every review mentions the name of the movie at least once. We can use that fact to find only the reviews for our target movie.

Normally producers and directors would stop this kind of garbage from getting published. <u>Truncated</u> is amazing in that it got past those hurdles.

- We can look for this within the string.

The verbose way to find array elements, using "each"

You can call the include? method on any instance of the String class to determine if it includes a substring (which you pass as an argument). Remember, by convention, methods that end in ? return a boolean value. The include? method will return true if the string contains the specified substring, and false if it doesn't.

```
my_string = "I like apples, bananas, and oranges"
puts my_string.include?("bananas")
puts my_string.include?("elephants")
false
```

It doesn't matter if the substring you're looking for is at the beginning of the string, at the end, or somewhere in the middle; include? will find it.

So, here's one way you could select only the relevant reviews, using the include? method and the other techniques we've learned so far...

```
(lines = [])
 Our old code to read ≥ File.open("reviews.txt") do |review file|
  the file contents. ( lines = review_file.readlines end
                                       Remember to create the
                from the file.
                  if line.include?("Truncated")
                    relevant lines << line 🤶
                  end
                                           Add the current line to
                end
                                           the array of reviews.
                puts relevant lines
                    Normally producers and directors would stop this kind of
                    garbage from getting published. Truncated is amazing in that
                    it got past those hurdles.
                        --Joseph Goldstein, "Truncated: Awful", New York Minute
   Review for other -
                    Truncated is funny - it can't be categorized as comedy,
   movie removed!
                    romance, or horror, because none of those genres would want
                    to be associated with it.
                        --Liz Smith, "Truncated Disappoints", Chicago Some-Times
                    I'm pretty sure this was shot on a mobile phone. Truncated
                    is astounding in its disregard for filmmaking aesthetics.
                        --Bill Mosher, "Don't See Truncated", Topeka Obscurant
```

Introducing a faster method...

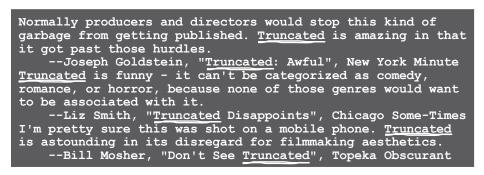
But actually, Ruby offers a much quicker way to do this. The find_all method uses a block to run a test against each element in an array. It returns a new array that contains only the elements for which the test returned a true value.

We can use the find_all method to achieve the same result, by calling include? in its block:

```
lines = []
File.open("reviews.txt") do |review_file|
    lines = review_file.readlines
end
relevant_lines = lines.find_all { |line| line.include?("Truncated") }
```

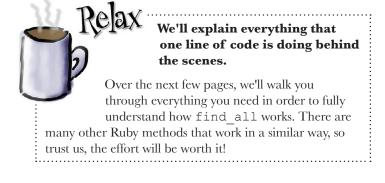
This shortened code works just as well: only lines that include the substring "Truncated" are copied to the new array!

```
puts relevant_lines
```



Replacing six lines of code with a single line... Not bad, huh?

Uh, oh. Did we just blow your mind again?



Blocks have a return value

We just saw the find_all method. You pass it a block with selection logic, and find_all finds only the elements in an array that match the block's criteria.

lines.find_all { |line| line.include?("Truncated") }

By "elements that match the block's criteria", we mean elements for which the block *returns* a true value. The find_all method uses the *return value of the block* to determine which elements to keep, and which to discard.

As we've progressed, you've probably noticed a few similarities between blocks and methods...

Methods:

- Accept parameters
- Have a body that holds Ruby expressions
- Return a value

- Blocks:
 - Accept parameters
 - Have a body that holds Ruby expressions
 - Return a value <

Wait, what? Do they?

That's right, just like methods, Ruby blocks return the value of the last expression they contain! It's returned to the method as the result of the yield keyword.

We can create a simple method that shows this in action, and then call it with different blocks to see their return values:

def print_block_result
 block_result = yield Assigns the result of the
 puts block_result
end
print_block_result { 1 + 1 }
print_block_result do
 "I'm not the last expression, so I'm not the return value."
 "I'm the result!"
end

print_block_result { "I hated Truncated".include?("Truncated") }

2 I'm the result! true

Blocks have a return value (cont.)

The method isn't limited to *printing* the block return value, of course. It can also do math with it:

```
def triple_block_result
   puts 3 * yield
end
triple_block_result { 2 }
triple_block_result { 5 }
15
```

...Or use it in a string:

```
def greet
  puts "Hello, #{yield}!"
end
greet { "Liz" } Hello, Liz!
```

...Or use it in a conditional:

```
def alert_if_true
    if yield
        puts "Block returned true!"
    else
        puts "Block returned false."
    end
end
alert_if_true { 2 + 2 == 5 }
    alert_if_true { 2 > 1 }
    Block returned true!
```

Up next, we'll take a detailed look at how find_all uses the block's return value to give you just the array elements you want.



We say that blocks have a "return value", but that doesn't mean you should use the return keyword.

Using the return *keyword within a* block isn't a syntax error, but we don't recommend it. Within a block body, the return keyword returns <u>from the method where the block is</u> <u>being defined</u>, not the block itself. It's very unlikely that this is what you want to do.

```
def print_block_value
   puts yield
end
def other_method
   print_block_value { return 1 + 1 }
end
```

other_method

The above code <u>won't print anything</u>, because other method exits <u>as the block is being defined</u>.

If you change the block to simply use its last expression as a return value, then everything works as expected:

def other_meth print_block_ end	od value { 1 + 1 }
other_method	2

bumb Questions

Q: Do all blocks return a value?

A: Yes! They return the result of the last expression in the block body.

Q: If that's true, then why didn't we learn about this sooner?

A: We haven't needed to. A block may return a value, but the associated method doesn't have to use it. The each method, for example, ignores the values returned from its block.

Q: Can I pass parameters to a block and use its return value?

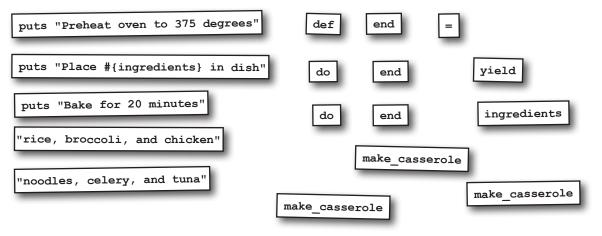
A: Yes! You can pass parameters, use the return value, do both, or do neither; it's up to you.

```
def one_two
  result = yield(1, 2)
  puts result
end
one_two do |param1, param2|
  param1 + param2
end
```



Code Magnets

A Ruby program is all scrambled up on the fridge. Can you reconstruct the code snippets so that they produce the given output?



Output:

File Edit Window Help
Preheat oven to 375 degrees
Place noodles, celery, and tuna in dish
Bake for 20 minutes
Preheat oven to 375 degrees
Place rice, broccoli, and chicken in dish
Bake for 20 minutes



Code Magnets Solution

A Ruby program is all scrambled up on the fridge. Can you reconstruct the code snippets so that they produce the given output?

def	make_casserole	make_casserole do
	puts "Preheat oven to 375 degrees"	"noodles, celery, and tuna"
	ingredients = yield	end
	<pre>puts "Place #{ingredients} in dish"</pre>	_
	puts "Bake for 20 minutes"	make_casserole do
end		"rice, broccoli, and chicken"
_		end
	Output:	

File Edit Window Help
Preheat oven to 375 degrees
Place noodles, celery, and tuna in dish
Bake for 20 minutes
Preheat oven to 375 degrees
Place rice, broccoli, and chicken in dish
Bake for 20 minutes

How the method uses a block return value

We're close to deciphering how this snippet of code works:

lines.find all { |line| line.include?("Truncated") }

The last step is understanding the find all method. It passes each element in an array to a block, and builds a new array including only the elements for which the block returns a true value.

> p [1, 2, 3, 4, 5].find all { |number| number.even? } p [1, 2, 3, 4, 5].find all { |number| number.odd? }



["a",

"b",

"c"

You can think of the values the block returns as a set of *instructions* for the method. The find all method's job is to keep some array elements and discard others. But it relies on the block's return value to tell it which elements to keep.

All that matters in this selection process is the block's return value. The block body doesn't even have to use the parameter with the current array element (although in most practical programs, it will). If the block returns true for everything, all the array elements will be included...

p ['a', 'b', 'c'].find all { |item| true }

...If it returns false for everything, none of them will be.

p ['a', 'b', 'c'].find all { |item| false }

Think of block return values as instructions from the block to the method.

```
[]
                                                             Create a new array to hold
If we were to write our own version
                                                             the elements for which the
                                                           block returns "true".
of find all, it might look like this:
                                   class Array
                                     def find all
                                                                    - Process each element.
                                       matching items = []
                                       self.each do |item| <
                                         result is "true" ...
                                            matching items << item 🔶
                                          end
                                                                        Add it to the array of
                                       end
                                                                        matching elements.
                                       matching items
                                     end
                                   end
                                            The old code:
                                                         relevant lines = []
If this code looks familiar, it should. It's a more
                                                         lines.each do |line|
generalized version of our earlier code to find
                                                            if line.include?("Truncated")
lines that were relevant to our target movie!
                                                              relevant lines << line
                                                            end
                                                          end
                                                          puts relevant lines
```

Putting it all together

Now that we know how the find_all method works, we're really close to understanding this code.

```
lines = []
File.open("reviews.txt") do |review_file|
    lines = review_file.readlines
end
relevant lines = lines.find all { |line| line.include?("Truncated") }
```

Here's what we've learned (not necessarily in order):

```
Result will be used as
     The last expression in a block becomes its return value.
•
                                                                              block return value.
                                    lines.find all { |line| line.include?("Truncated") }
•
     The include? method returns true if the string contains the
     specified substring, and false if it doesn't.
                                    lines.find all { |line| line.include?("Truncated") }

    Returns true if line
contains "Truncated".

     The find all method passes each element in an array to a
•
     block, and builds a new array including only the elements for
     which the block returns a true value.
                                    lines.find all { |line| line.include?("Truncated") }

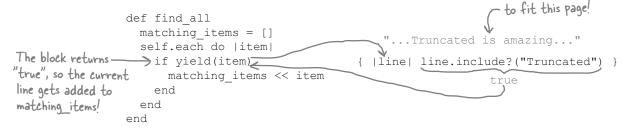
    Result will be an array with all the elements of "lines"
that contain string "Truncated".
```

Let's look inside the find_all method and the block as they process the first few lines of the file, to see what they're doing...

A closer look at the block return values

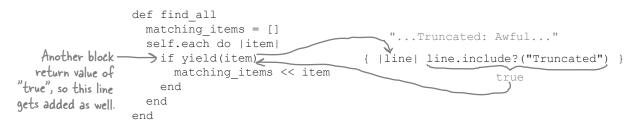
1

The find_all method passes the first line from the file to the block, which receives it in the line parameter. The block tests whether line includes the string "Truncated". It does, so the return value of the block is true. Back in the method, the line gets added to the array of matching items.



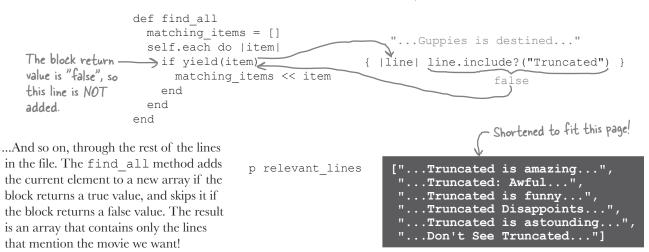
2

The find_all method passes the second line from the file to the block. Again, the line block parameter includes the string "Truncated", so the return value of the block is again true. Back in the method, this line also gets added to the array of matching items.





The *third* line from the file *doesn't* include the string "Truncated", so the return value of the block is false. This line is *not* added to the array.

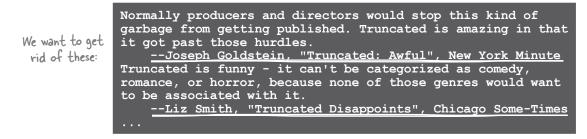


Eliminating elements we don't want, with a block

Using the find_all method, we've successfully found all the reviews for our target movie, and placed them in the relevant_lines array. We can check another requirement off our list!

Constant Sector	V	Get the file contents.
	\square	Find reviews for the current movie.
		Discard reviewer bylines.

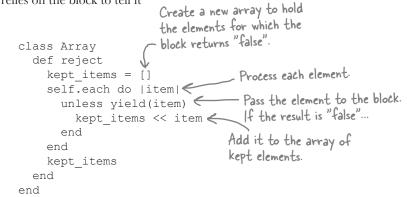
Our next requirement is to discard the reviewer bylines, because we're only interested in retrieving adjectives from the main text of each review.



Fortunately, they're clearly marked. Each one starts with the characters "--", so it should be easy to use the include? method to determine if a string contains a byline.

Before, we used the find_all method to *keep* lines that included a particular string. The reject method is basically the opposite of find_all - it passes elements from an array to a block, and *rejects* an element if the block returns a true value. If find_all relies on the block to tell it which items to *keep*, reject relies on the block to tell it which items to *keep*.

If we were to implement our own version of reject, it would look very similar to find all:



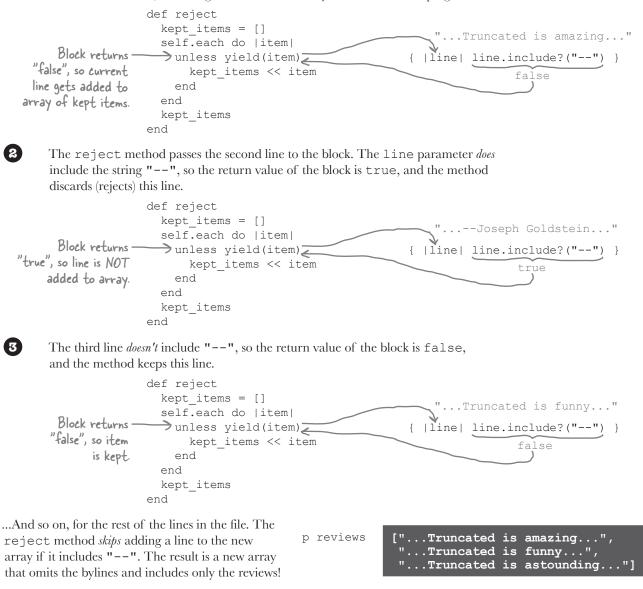
The return values for "reject"

So reject works just like find_all, except that instead of *keeping* elements that the block returns a true value for, it *rejects* them. Using reject, it should be easy to get rid of the bylines!

```
reviews = relevant lines.reject { |line| line.include?("--") }
```

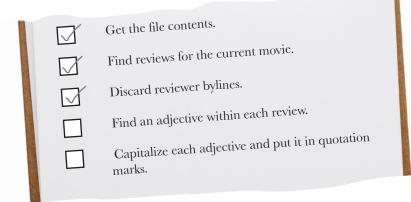


The reject method passes the first line from the file to the block. The line block parameter does *not* include the string "--", so the return value of the block is false. Back in the method, this line gets added to the array of items we're keeping.



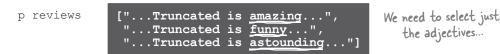
Breaking a string into an array of words

We've discarded the reviewer bylines, leaving us with an array containing only the text of each review. That's another requirement down! Two to go...



For our next requirement, we're going to need a couple new methods. They don't take blocks at all, but they *are* super-useful.

We need to find an adjective in each review:



If you look above, you'll notice a pattern... The adjective we want always seems to follow the word "is".

So, we need to get one word that follows another word... What we have right now are *strings*. How can we convert those to *words*?

Strings have a split instance method that you can call to split them into an array of substrings.

p "1-800-555-0199".split("-")	["1", "800", "555", "0199"]
p "his/her".split("/")	["his", "her"]
<pre>p "apple, avocado, anvil".split(", ")</pre>	["apple", "avocado", "anvil"]

The argument to split is the *separator*: one or more characters that separate the string into sections.

What separates words in the English language? A space! If we pass " " (a space character) to split, we'll get an array back. Let's try it with our first review.

```
string = reviews.first
words = string.split(" ")
p words
```

["Normally", "producers", "and", "directors",
"would", "stop", "this", "kind", "of", "garbage",
"from", "getting", "published.", "Truncated", "is",
"amazing", "in", "that", "it", "got", "past",
"those", "hurdles."]

There you have it - an array of words!

Finding the index of an array element

The split method converted our review string into an array of words. Now, we need to find the word "is" within that array. Again, Ruby has a method ready to go for us. If you pass an argument to the find_index method, it will find us the first index where that element occurs in the array.

```
p ["1", "800", "555", "0199"].find_index("800")
p ["his", "her"].find_index("his")
p ["apple", "avocado", "anvil"].find_index("anvil")
```

```
1
0
2
```

Using find_index, let's write a method that will split a string into an array of words, find the index of the word "is", and return the word that comes *after* that.

```
def find_adjective(string)

words = string.split(" ")

index = words.find_index("is") Find the array index of "is".

words[index + 1]

end

Find the word AFTER

"is", and return it.
```

We can easily test our method out on one of our reviews...

adjective = find_adjective(reviews.first)

```
amazing
```

There's our adjective! That only takes care of one review, though. Next, we need to process *all* the reviews, and create an array of the adjectives we find. With the each method, that's easy enough to do.

```
Create a new array to

adjectives = []

reviews.each do |review| 	For each review in the array...

adjectives << find_adjective (review)

end

puts adjectives

amazing

funny

astounding
```

Now we have an array of adjectives, one for each review!

Would you believe there's an even *easier* way to create an array of adjectives based on the array of reviews, though?

Making one array that's based on another, the hard way

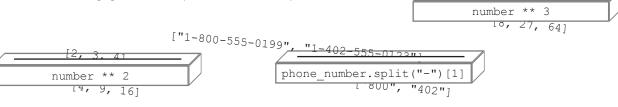
We had no problem looping through our array of reviews to build up an array of adjectives using each and our new find adjective method.

But creating a new array based on the contents of another array is a really common operation, that requires similar code each time. Some examples:

numbers = [2, 3, 4]numbers = [2, 3, 4]squares = [] - Make an array Loop cubes = [] - Make an array LOOP to hold results. through to hold results. numbers.each do |number| numbers.each do |number| through source squares << number ** 2 cubes << number ** 3 source array. end end array. Perform an operation, and Perform an operation, and p squares copy result to results array. p cubes copy result to results array. [4, 9, 16] [8, 27, 64] phone numbers = ["1-800-555-0199", "1-402-555-0123"] area_codes = [] - Make an array Loop through _ source array. to hold results. phone_numbers.each do |phone number| 🧲 area codes << phone number.split("-")[1] end Perform an operation, and copy result to results array. p area codes ["800", "402"]

In each of these examples, we have to set up a new array to hold the results, loop through the original array and apply some logic to each of its members, and add the result to the new array. (Just like in our adjective finder code.) It's a bit repetitive...

Wouldn't it be great if there were some sort of magic processor for arrays? You drop in your array, it runs some (interchangeable) logic on its elements, and out pops a new array with the elements you need!



Making one array that's based on another, using "map"

Ruby has just the magic array processor we're looking for: the map method. The map method takes each element of an array, passes it to a block, and builds a new array out of the values the block returns.

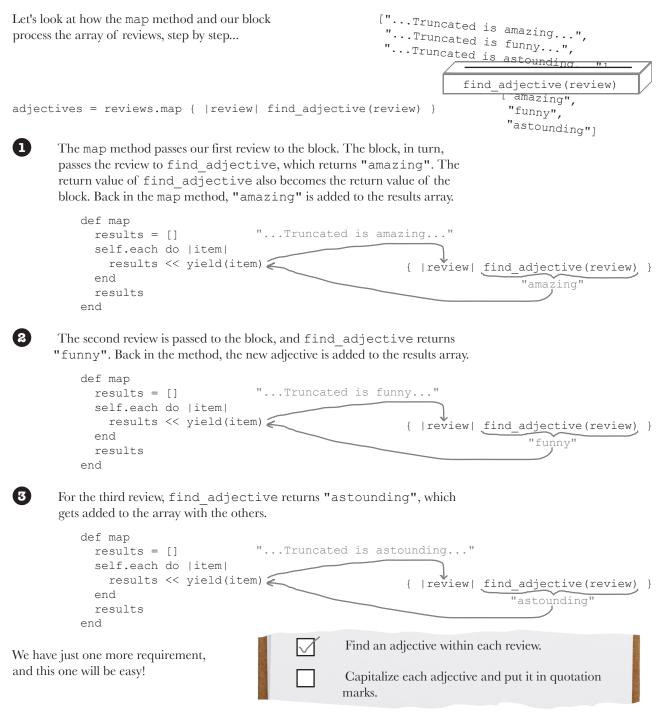
```
No need to create the
result arrays beforehand -
"map" creates them for us!
squares = [2, 3, 4].map { |number| number ** 2 }
Cubes = [2, 3, 4].map { |number| number ** 2 }
area_codes = ['1-800-555-0199', '1-402-555-0123'].map do |phone|
phone.split("-")[1] 
Make a new array with
end
p squares, cubes, area_codes
[4, 9, 16]
[8, 27, 64]
["800", "402"]
```

The map method is similar to find_all and reject, in that it processes each element in an array. But find_all and reject use the block's return value to decide whether to copy *the original element* from the old array to the new one. The map method adds the block's *return value itself* to the new array.

If we were to code our own version of map, it might look like this:

Make a new array to hold the ss Array block return values. ef map Loop through results = [] each element. class Array def map the return value to the new array. end results 🤶 Return the array of end end block return values. The map method can shorten our code to gather adjectives down to a single line! An array with all the return values from find_adjective. adjectives = reviews.map { |review| find adjective(review) } Call our method. Its return value will be the The return value of map is an array with all the values the block returned: return value of the block. ["amazing", "funny", "astounding"] p adjectives

Making one array that's based on another, using "map" (cont.)

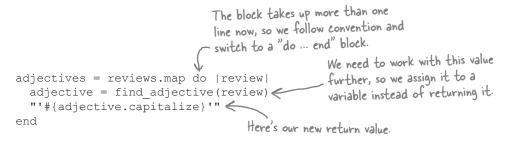


Some additional logic in the "map" block body

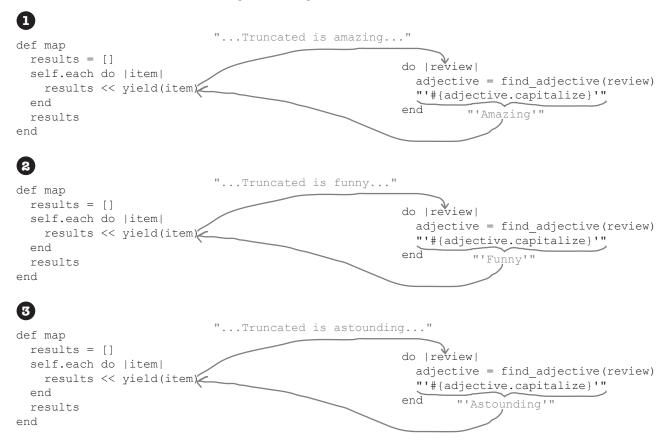
We're already using map to find the adjectives for each review:

adjectives = reviews.map { |review| find_adjective(review) }

We can just add code to capitalize the adjective and enclose it in quotation marks to the block, right after the call to our find adjective method.

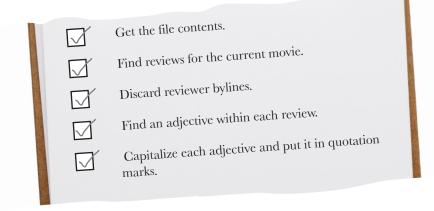


Here are the new return values that this updated code produces:



The finished product

That's our last requirement. Congratulations, we're done!



You've successfully learned to use block return values to find elements you want within an array, reject elements you don't want, and even to use an algorithm to create an entirely new array!

Processing a complex text file like this would take dozens of lines of code in other languages, with lots of repetition. The find_all, reject, and map methods handled all of that for you! They're not the easiest methods to learn to use, but now that you have, you've got powerful new tools at your disposal!

We'll call this method below, to def find adjective (string) find adjectives within each review. Here's our complete code listing: words = string.split(" ") - Break the string into an array of words words [index + 1] - Return the word following "is". end lines = [] - We need to create this variable outside the block. File.open ("reviews.txt") do |review_file| - Open the file, and automatically close it when we're done. lines = review file.readlines 🧲 end Read every line in the file into an array. Find lines that include reviews = relevant lines.reject { |line| line.include?("--") } Exclude reviewer bylines. - Process each review. adjectives = reviews.map do |review| < adjective = find adjective (review) - Find the adjective. " '# {adjective.capitalize} ' " Return the adjective, capitalized and surrounded end by quotes. The critics agree, Truncated is: puts "The critics agree, Truncated is:" 'Amazing' puts adjectives 'Funny' 'Astounding'



Open a new terminal or command prompt, type "irb" and hit the Enter/Return key. For each of the Ruby expressions below, write your guess for what the result will be on the line next to it. Then try typing the expression into irb, and hit Enter. See if your guess matches what irb returns!

[1, 2,	з,	<pre>4].find_all { number number.odd? }</pre>	
[1, 2,	3,	<pre>4].find_all { number true }</pre>	
[1, 2,	3,	<pre>4].find_all { number false }</pre>	
[1, 2,	3,	<pre>4].find { number number.even? }</pre>	
[1, 2,	3,	<pre>4].reject { number number.odd? }</pre>	
[1, 2,	3,	<pre>4].all? { number number.odd? }</pre>	
[1, 2,	3,	<pre>4].any? { number number.odd? }</pre>	
[1, 2,	3,	4].none? { number number > 4 }	
[1, 2,	3,	<pre>4].count { number number.odd? }</pre>	
[1, 2,	, 3,	<pre>4].partition { number number.odd? }</pre>	

['\$',	'\$\$',	<pre>'\$\$\$'].map { string string.length }</pre>	
['\$',	'\$\$',	'\$\$\$'].max_by { string string.length }	
['\$',	'\$\$',	'\$\$\$'].min_by { string string.length }	



Open a new terminal or command prompt, type "irb" and hit the Enter/Return key. For each of the Ruby expressions below, write your guess for what the result will be on the line next to it. Then try typing the expression into irb, and hit Enter. See if your guess matches what irb returns!

[1, 2	2, 3	3,	41.find	all	{	number	number.	.odd?	}

- [1, 2, 3, 4].find all { |number| true }
- [1, 2, 3, 4].find_all { |number| false }
- [1, 2, 3, 4].find { |number| number.even? }
- [1, 2, 3, 4].reject { |number| number.odd? }
- [1, 2, 3, 4].all? { |number| number.odd? }
- [1, 2, 3, 4].any? { |number| number.odd? }
- [1, 2, 3, 4].none? { |number | number > 4 }
- [1, 2, 3, 4].count { |number| number.odd? }
- [1, 2, 3, 4].partition { |number| number.odd? }

An array of all values for which [1, 3] - the block returns "true". If it always returns "true", [1, 2, 3, 4] - all values get included. If it NEVER returns true, [] - NO values are included. "find" returns the FIRST value for 2 - which the block returns "true". "all?" returns true if the block false - returned true for ALL elements. "any?" returns true if the block true - returned true for ANY elements. "none?" returns true if the block true - returned FALSE for all elements. The number of elements for which 2 - the block returned "true".

[[], 3], [2, 4]]

Two arrays, the first with all the elements where the block returned TRUE, the second with all the elements where it returned FALSE.

['\$', '\$\$', '\$\$\$'].map { |string| string.length }
['\$', '\$\$', '\$\$\$'].max_by { |string| string.length }
['\$', '\$\$', '\$\$\$'].min_by { |string| string.length }

An array with all the [1, 2, 3] — values the block returns. The element for which the block "fif" — returned the LARGEST value. The element for which the block "f" — returned the SMALLEST value.



Your Ruby Toolbox

That's it for Chapter 6! You've added block return values to your tool box.

11 1 Arrays An Blocks Block Return Values The value of the last expression in a block's body is returned to the method, as the value of the yield keyword. Methods can use the block return value to find elements in a collection, decide how to sort elements, and more.

BULLET POINTS

- If you pass a block to File.open, it will yield the file to the block so you can do whatever you need with it. When the block ends, the file will automatically be closed.
- Strings have an instance method called include?, which takes a substring as an argument. It will return true if the string includes the substring, false if not.
- When you need to find all elements of an array that meet some criteria, you can use the find_all method. It passes each element of the array to a block, and will return a new array with all the elements for which the block returned a true value.
- The reject method works just like find_all, except that it rejects array elements for which a block returns a true value.
- The split method on strings takes a separator as an argument. It finds each instance of the separator within the string, and returns an array with all of the substrings that were between each separator.
- The find_index method searches for the first occurrence of an element within an array, and returns its index.
- The map method takes each element of an array, passes it to a block, and builds a new array out of the values the block returns.

page goal header

7 hashes

Labelling Data



Throwing things in piles is fine, until you need to find something

again. You've already seen how to create a collection of objects using an *array*. You've seen how to process *each item* in an array, and how to *find items* you want. In both cases, you start at the beginning of the array, and *look through Every. Single. Object.*

You've also seen methods that take big collections of parameters. You've seen the problems this causes: method calls require a big, *confusing collection of arguments* that you have to remember the exact order for.

What if there was a kind of collection where *all the data had labels* on it? You could *quickly find the elements* you needed! In this chapter, we'll learn about Ruby **hashes**, which do just that.

Counting votes

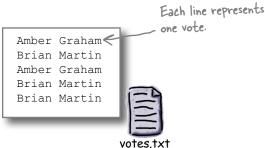
A seat on the Sleepy Creek County School Board is up for grabs this year, and polls have been showing the election to be really close. Now that it's election night, the candidates are excitedly watching the votes roll in.



The electronic voting machines in use this year record the votes to text files, one vote per line. (Budgets are tight, so the city council chose the cheap voting machine vendor.)

Here's a file with all the votes for District A:

We need to process each line of the file, and tally the total number of times each name occurs. The name with the most votes will be our winner!



The development team's first order of business is to read the contents of the "votes.txt" file. That part is easy; it's just like the code we used to read the movie reviews file back in Chapter 6.

Create a variable that will still be accessible after the block. File.open("votes.txt") do |file| Open the file, and pass lines = file.readlines it to the block. end

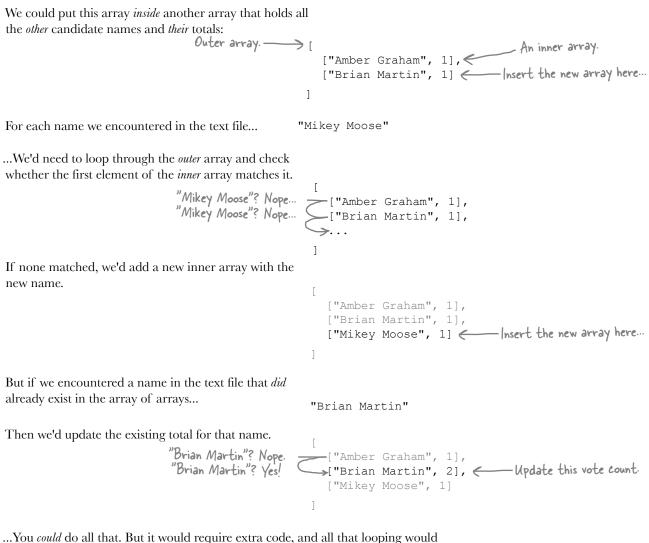
Now, we need to get the name from each line of the file, and increment a tally of the number of times that name has occurred.

An array of arrays... is not ideal

But how do we keep track of all those names *and* associate a vote total with each of them? We'll show you two ways. The first approach uses arrays, which we already know about from Chapter 5. Then we'll show you a second way using a new data structure, *hashes*.

If all we had to work with were arrays, we might build an *array of arrays* to hold everything. That's right, Ruby arrays can hold any object, including *other arrays*. So we could create an array with the candidate's name, and the number of votes we've counted for it:

["Brian Martin", 1]



take a long time when processing large lists. As usual, Ruby has a better way.

Hashes

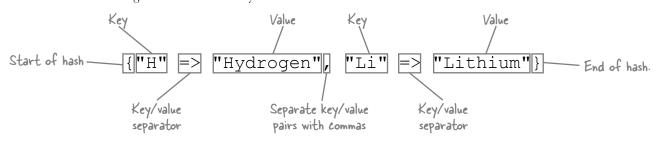
The problem with storing the vote tally for each candidate in an array is the inefficiency of looking them up again later. For each name we want to find, we have to search through *all* the others.

Putting data in an array is like stacking it in a big pile; you can get particular items back out, but you'll have to search through everything to find them. "Mikey Moose"? Nope... "Mikey Moose"? Nope... "Mikey Moose"? ["Brian Martin", 5], "Mikey Moose"?]

Ruby has another way of storing collections of data... *hashes*. A **hash** is a collection where each value is accessed via a *key*. Keys are an easy way to get data back out of your hash. It's like having neatly labelled file folders instead of a messy pile.



Just like with arrays, you can create a new hash and add some data to it at the same time using a hash literal. The syntax looks like this:



Those => symbols show which key points to which value. They look a bit like a rocket, so they are sometimes called "hash rockets".

We can assign a new hash to a variable:

Then, we can access values from that hash using the keys we set up for them. Whereas hash literals use *curly braces*, you use *square brackets* to access individual values. It looks just like the syntax to access values from an array, except you place the hash key within the brackets instead of a numeric index. Use a hash key here, and you'll get the corresponding value. puts elements ["Li"] puts elements ["H"] Hydrogen

Hashes (cont.)

We can also add new keys and values to an existing hash. Again, the syntax looks a lot like the syntax to assign to an array element:

Whereas an array can only use *integers* as indexes, a hash can use *any object* as a key. That includes numbers, strings, and symbols.

```
mush = {1 => "one", "two" => 2, :three => 3.0}
p mush[:three] 3.0
p mush[1] "one"
p mush["two"] 2
```

Although arrays and hashes have major differences, there are enough similarities that it's worth taking a moment to compare them...

Arrays:

- Grow and shrink as needed
- Can hold any object, even hashes or other arrays
- Can hold instances of more than one class at the same time
- Literals surrounded by square brackets
- Elements accessed by specifying index within *square brackets*
- Only integers can be used as indexes

[2.99, 25.00, 9.99]

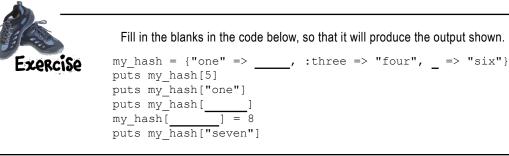
10

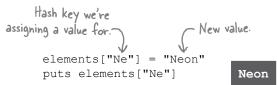
• Index of an element is determined by position within array

Hashes:

•

- Grow and shrink as needed
- Can hold any object, even arrays or other hashes
- Can hold instances of more than one class at the same time
- Literals surrounded by curly braces
- Values accessed by specifying key within *square brackets*
- Any object can be used as a key
- Keys not calculated; key must be specified whenever a value is added





Output:

six

two

8

four

	Fill in the blanks in the code below, so that it will produce the output shown.	
Exercise Solution	<pre>my_hash = {"one" => "two", :three => "four", 5 => "six"} puts my_hash[5] puts my_hash["one"] puts my_hash[:three_] my_hash["seven"] = 8 puts my_hash["seven"]</pre>	Output: six two four 8

Hashes are objects

We've been hearing over and over that everything in Ruby is an object. We saw that arrays are objects, and it probably won't surprise you to learn that hashes are objects, too.

protons = $\{"H" \Rightarrow 1,$	"Li" => 3,	"Ne" => 10}	
puts protons.class			Hash

And, like most Ruby objects, hashes have lots of useful instance methods. Here's a sampling...

They have the methods that you expect every Ruby object to have, like inspect:	<pre>puts protons.inspect {"H"=>1, "Li"=>3, "Ne"=>10}</pre>
The length method lets you determine how many key/value pairs the hash holds:	puts protons.length 3
There are methods to quickly test whether the hash includes particular keys or values:	<pre>puts protons.has_key?("Ne") true puts protons.has_value?(3) true</pre>
There are methods that will give you an array with all the keys, or all the values:	<pre>p protons.keys ["H", "Li", "Ne"] p protons.values [1, 3, 10]</pre>

And, as with arrays, there are methods that will let you use a block to iterate over the hash's contents. The each method, for example, takes a block with *two* parameters, one for the key and one for the value. (More about each in a few pages.)

protons.each do |element, count|
 puts "#{element}: #{count}"
end





Open a new terminal or command prompt, type "irb" and hit the Enter/Return key. For each of the Ruby expressions below, write your guess for what the result will be on the line next to it. Then try typing the expression into irb, and hit Enter. See if your guess matches what irb returns!

protons = { "He" => 2 }	
protons["He"]	
<pre>protons["C"] = 6</pre>	
protons["C"]	
<pre>protons.has_key?("C")</pre>	
protons.has_value?(119)	
protons.keys	
protons.values	
<pre>protons.merge({ "C" => 0,</pre>	, "Uh" => 147.2 })

there are no Dumb Questions

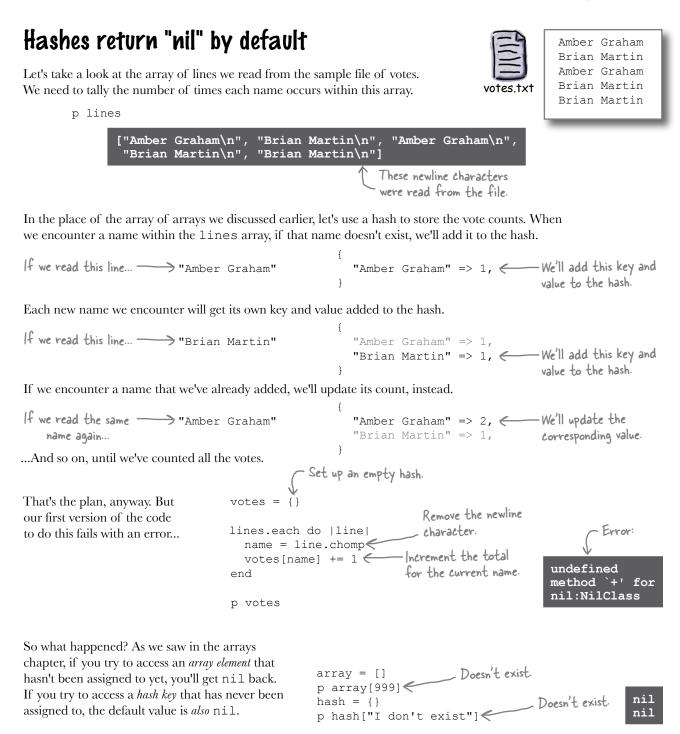
Q: Why do they call it a "hash"?

A: Frankly, it's not the best possible name. Other languages refer to this kind of structure as "maps", "dictionaries", or "associative arrays" (because keys are associated with values). In Ruby, it's called a "hash" because an algorithm called a *hash table* is used to quickly look up keys within the hash. The details of that algorithm are beyond the scope of this book, but you can visit your favorite search engine to learn more.



Open a new terminal or command prompt, type "irb" and hit the Enter/Return key. For each of the Ruby expressions below, write your guess for what the result will be on the line next to it. Then try typing the expression into irb, and hit Enter. See if your guess matches what irb returns!

01	Result of an assignment statement, as always, is the
protons = { "He" => 2 }	{"He"=>2} value that was assigned.
protons["He"]	Provide the key, get the corresponding value.
<pre>protons["C"] = 6</pre>	The value that was assigned.
protons["C"]	Retrieving the value we just assigned from the hash.
<pre>protons.has_key?("C")</pre>	"true" because the hash includes the given key.
protons.has_value?(119)	"false" because no key in the false hash has the given value.
protons.keys	An array containing <i>C</i> "He", "C"] every key in the hash.
protons.values	An array containing every value in the hash.
	If a key in the new hash already exist, already exists in the old hash, it just gets the old value is overridden. added.
<pre>protons.merge({ "C" => 0,</pre>	



When we try to access the votes for a candidate name that has never been assigned to, we get nil back. And trying to add to nil produces an error.

Hashes return "nil" by default (cont.)

The first time we encounter a candidate's name, instead of getting a vote tally back from the hash, we get nil. This results in an error when we try to add to it.

To fix this, we can test whether the value for the current hash key is nil. If it's not, then we can safely increment whatever number is there. But if it *is* nil, then we'll need to set up an initial value (a tally of 1) for that key.

```
lines.each do |line|
  name = line.chomp
  votes[name] += 1
end
```

undefined method `+' for nil:NilClass

And in the output, we see the populated hash. Our code is working!

nil (and only nil) is "falsy"

There's a small improvement to be made, though... That conditional is a little ugly. if votes[name] != nil

We can clean that up by taking advantage of the fact that *any* Ruby expression can be used in a conditional statement. Most of them will be treated as if they were a "true" value. (Rubyists often say these values are *"truthy"*.)

In fact, aside from the false boolean value, there is only *one* value that Ruby treats as if it was false: nil. (Rubyists often say that nil is *"falsy"*.)

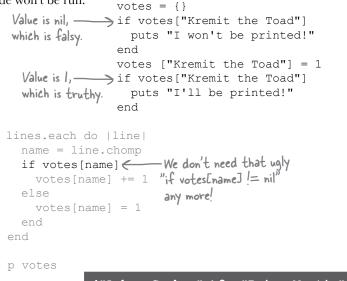
if false Actually false. puts "I won't be printed!" end if nil Falsy. puts "I won't, either!" end

nil (and only nil) is "falsy" (cont.)

Ruby treats nil like it's false to make it easier to test whether values have been assigned or not. For example, if you access a hash value within an if statement, the code within will be run if the value exists. If the value doesn't exist, the code won't be run.

We can make our conditional read a little better by changing it from "if votes [name] != nil" to just "if votes [name]".

Our code still works the same as before; it's just a bit cleaner looking. This may be a small victory now, but the average program has to test for the existence of objects a *lot*. Over time, this technique will save you many keystrokes!

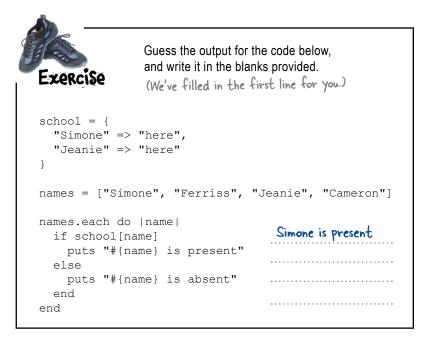


{"Amber Graham"=>2, "Brian Martin"=>3}



We mean it when we say that <u>only</u> nil is falsy.

Most values that are treated as falsy in some other languages, such as empty strings, empty arrays, and the number 0, are <u>truthy</u> in Ruby.



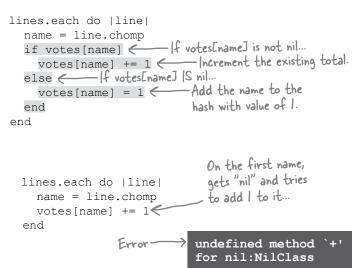
```
Guess the output for the code below,
                 and write it in the blanks provided.
Exercise
DOLUTION
school = \{
  "Simone" => "here",
  "Jeanie" => "here"
}
names = ["Simone", "Ferriss", "Jeanie", "Cameron"]
names.each do |name|
                                      Simone is present
  if school[name]
    puts "#{name} is present"
                                      Ferriss is absent
  else
                                      Jeanie is present
    puts "#{name} is absent"
  end
                                      Cameron is absent
end
```

A hash that returns something other than "nil" by default

votes = $\{\}$

A disproportionate amount of our code for tallying the votes lies in the if/else statement that checks whether a key exists within the hash...

And we *need* that if statement. Normally, when you try to access a hash key that hasn't had a value assigned yet, you get nil back. We'd get an error the first time we tried to add to the tally for a key that didn't yet exist (because you can't add to nil).



But... what if, when we tried to access a hash key that hasn't been assigned to yet, we got a different value instead of nil? One that we *can* increment? Let's find out how to make that happen...

When we access a value

that's been assigned to,

A hash that returns something other than "nil" by default (cont.)

votes = Hash.new

votes["Amber Graham"] = 1

Instead of using a hash literal ({ }), you can also call Hash.new to create new hashes. Without any arguments, Hash.new works just like { }, giving you a hash that returns nil for unassigned keys.

But when you call Hash.new and pass an object as an argument, that argument becomes that hash's default object. Anytime you access a key in that hash that hasn't been assigned to yet, instead of nil, you'll get the default object you specified ...

Let's use a hash default object to shorten up our vote counting code...

If we create our hash with Hash.new(0), it will return the default object (0) when we try to access the vote tally for any key that hasn't been assigned to yet. That 0 value gets incremented to 1, then 2, and so on as the same name is encountered again and again.

statement entirely!

we get that value back. p votes["Amber Graham"] < -When we access a value p votes["Brian Martin"] < that HASN'T been assigned to, we get "nil". 1 nil Create a new hash with a default object of "O' When we access a value that's been assigned to, votes = Hash.new(0)we get that value back. votes["Amber Graham"] = 1 p votes["Amber Graham"] < When we access a value p votes["Brian Martin"] <

Create a new hash.

1 0





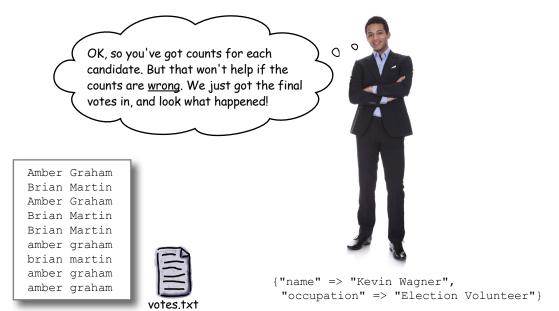
Using anything other than a number as a hash default object may cause bugs!

We'll cover ways to safely use other objects in Chapter 8. Until then, don't use anything other than a number as a default!

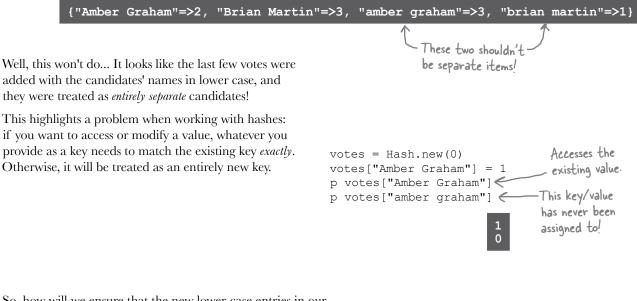
```
lines = []
                             File.open("votes.txt") do |file|
                               lines = file.readlines
                                                             Create a new hash with
                             end
                                                            a default object of "O".
                             votes = Hash.new(0) <
                                                           Increment whatever value is returned:
                             lines.each do |line|
                                                           "O" if the key has never been updated,
                               name = line.chomp
                                                            or the current tally otherwise.
We can get rid of the if
                               votes[name] += 1
                             end
And as you can see from the
                             p votes
                                           {"Amber Graham"=>2, "Brian Martin"=>3}
output, the code still works.
```

237 you are here ▶

Normalizing hash keys



Here's what we get if we run this new file through our existing code:



So, how will we ensure that the new lower-case entries in our text file get matched with the capitalized entries? We need to *normalize* the input: we need one standard way of representing candidates' names, and we need to use *that* for our hash keys.

Normalizing hash keys (cont.)

Fortunately, in this case, normalizing the candidate names is really easy. We'll add one line of code to ensure the case on each name matches prior to storing it in the hash.

<pre>lines = [] File.open("votes.txt") do file lines = file.readlines end</pre>
votes = {}
<pre>lines.each do line name = line.chomp name.upcase! Change the name to ALL CAPS if votes[name] before adding it to the hash (or votes[name] += 1 else votes[name] = 1 end end</pre>
p votes
{"AMBER GRAHAM"=>5, "BRIAN MARTIN"=>4}
We have our winner!
We have our winner! You also need to normalize the keys when <u>accessing</u> values.
You also need to normalize the keys when

And in the output we see the updated contents of our hash: votes from the lowercase entries have been added to the totals for the capitalized entries. Our counts are fixed!

.....

Hashes and "each"

We've processed the lines in the sample file, and built a hash with the total number of votes:

p votes {"AMBER GRAHAM"=>5, "BRIAN MARTIN"=>4}

1

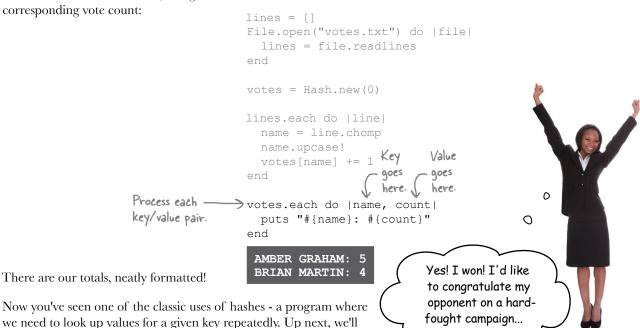
2

It would be far better, though, if we could print one line for each candidate name, together with their vote count.

As we saw back in Chapter 5, arrays have an each method that takes a block with a single parameter. The each method passes each element of the array to the block for processing, one at a time. Hashes also have an each method, that works in about the same way. The only difference is that on hashes, each expects a block with *two* parameters, one for the key, and one for the corresponding value.

look at another common way to use hashes: as method arguments.

We can use each to print the name of each candidate in the votes hash, along with the corresponding vote count:



there are no Dumb Questions

What happens if I call each on a hash, but pass it a block with <u>one</u> parameter?

A: The each method for hashes allows that; it will pass the block a 2-element array with the key and value from each key/value pair in the hash. It's much more common to use blocks with two parameters, though.



Hash:

Nice to see you again, Array.

There's no need to be like that.

Tonight's talk: **An array and a hash** work out their differences.

Array:

I didn't really want to be here, but whatever, Hash.

Isn't there? I was doing a perfectly fine job storing everyone's collections, and then you come along, and developers everywhere are like, "Ooh! Why use an array when I can use a hash? Hashes are so cool!"

Well, I do have a certain glamor about me... But even I know there are still times when developers *should* use an array instead of a hash.

It's true; it's a lot of work keeping all of my elements where I can retrieve them quickly! It pays off if someone wants to retrieve a particular item from the middle of the collection, though. If they give me the correct key, I always know right where to find a value.

Yes, but the developer has to know the exact index where the data is stored, right? All those numbers are a pain to keep track of! But it's either that, or wait for the array to search through all its elements, one by one...

Agreed. Developers should know about both arrays *and* hashes, and pick the right one for their current task.

Darn right! Arrays are way more efficient than hashes! If you're happy retrieving elements in the same order you added them (say, with each), then you want an array, because you won't have to wait while a hash organizes your data for you.

Hey, we arrays can get data back too, you know.

But the point is, we *can* do it. And if you're just building a simple queue, we're still the better choice.

Fair enough.

A mess of method arguments

class Candidate

Suppose we're making an app to track basic information regarding candidates so voters can learn about them. We've created a Candidate class to keep all of a candidate's info in one convenient place. For convenience, we've set up an initialize method so that we can set all of an instance's attributes directly from a call to Candidate.new.

> - accessors. -Set up Candidate.new to

Candidate: Carl Barnes

Occupation: Attorney

Birthplace: Miami

Age: 49

Hobby:

Set up attribute

```
attr accessor :name, :age, :occupation, :hobby, :birthplace <
        def initialize(name, age, occupation, hobby, birthplace) <
 Use the
                                                                           take arguments.
          (self.name = name
parameters
          self.age = age
to set the .
          self.occupation = occupation
 object
          self.hobby = hobby
attributes. [self.birthplace = birthplace
        end
      end
```

Let's add some code following the class definition to create a Candidate instance, and print out its data.

```
def print summary(candidate)
  puts "Candidate: #{candidate.name}"
  puts "Age: #{candidate.age}"
  puts "Occupation: #{candidate.occupation}"
                                                                 We have to provide an
  puts "Hobby: #{candidate.hobby}"
                                                                 argument even if we're
  puts "Birthplace: #{candidate.birthplace}"
                                                               - not using it.
end
candidate = Candidate.new("Carl Barnes", 49, "Attorney", nil, "Miami")
print summary (candidate)
```

Our very first attempt at calling Candidate.new shows that its usage could be a lot smoother. We have to provide all the arguments whether we're going to use them or not.

We could just make the hobby parameter optional, if it didn't have the birthplace parameter following it ...

```
class Candidate
  attr accessor :name, :age, :occupation, :hobby, :birthplace
  def initialize(name, age, occupation, hobby = nil, birthplace)
                                                       - Provide a default value to make
  end
                                                        the parameter optional ...
end
```

Since birthplace is present, though, we get an error if we try to omit hobby...

```
Candidate.new("Carl Barnes", 49, "Attorney", , "Miami")
```

```
Error-
          syntax error, unexpected ',', expecting
```

A mess of method arguments (cont.)

We encounter another problem if we forget the order that method arguments should appear in...

```
candidate = Candidate.new("Amy Nguyen", 37, "Lacrosse", "Engineer", "Seattle")
print_summary(candidate)
Wait, what order do these go in?
```

It's becoming clear that there are some issues with using a long list of parameters for a method. The order is confusing, and it's hard to leave unwanted arguments off.

```
Whoops! We
got these two
backwards! Candidate: Amy Nguyen
Age: 37
Occupation: Lacrosse
Hobby: Engineer
Birthplace: Seattle
```

Using hashes as method parameters

Historically, Rubyists have dealt with these issues by using hashes as method parameters. Here's a simple area method that, instead of separate length and width parameters, accepts a single hash. [We realize this is a bit ugly. Over the next few pages, we'll show you some shortcuts to make hash parameters much more readable!]

```
Take one hash instead of multiple parameters.

def area (options)

options [:length] * options [:width]

end

Access values from the hash

instead of individual parameters.

puts area ({:length => 2, :width => 4})

Instead of passing multiple arguments, pass a

single hash with appropriate keys and values.
```

The convention in Ruby is to use symbols instead of strings for hash parameter keys, because looking up symbol keys is more efficient than looking up strings.

Using hash parameters offers several benefits over regular method parameters...

With regular parameters:

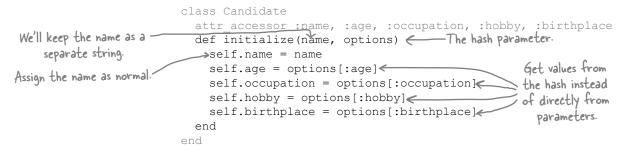
- Arguments must appear in exactly the right order
- Arguments can be hard to tell apart
- Required parameters have to appear before optional parameters

With hash parameters:

- Keys can appear in any order
- Keys act as "labels" for each value
- Can skip providing a value for any key you want

Hash parameters in our Candidate class

Here's a revision of our Candidate class's initialize method using a hash parameter.



We can now call Candidate.new by passing the name as a string, followed by a hash with the values for all the other Candidate attributes:

```
Now it's clear which attribute is which!
candidate = Candidate.new("Amy Nguyen",
  {:age => 37, :occupation => "Engineer", :hobby => "Lacrosse", :birthplace => "Seattle"})
p candidate
                   #<Candidate:0x007fbd7a02e858 @name="Amy Nguyen", @age=37,</pre>
                    @occupation="Engineer", @hobby="Lacrosse", @birthplace="Seattle">
                                  (No more switched attributes)
We can leave one or more of the hash keys off, if we want. The
attribute will just get assigned the hash default object, nil.
{:age => 49, :occupation => "Attorney", :birthplace => "Miami"})
p candidate
                #<Candidate:0x007f8aaa042a68 @name="Carl Barnes", @age=49,</pre>
                 @occupation="Attorney", @hobby=nil, @birthplace="Miami">
                                                     Omitted attributes default to nil
We can put the hash keys in any order we want:
candidate = Candidate.new("Amy Nguyen",
```

```
candidate conditate (new (new Nguyen ,
{:birthplace => "Seattle", :hobby => "Lacrosse", :occupation => "Engineer", :age => 37})
p candidate
#<Candidate:0x007f81a890e8c8 @name="Amy Nguyen", @age=37,
    @occupation="Engineer", @hobby="Lacrosse", @birthplace="Seattle">
```

Leave off the braces!

We'll admit that the method calls we've been showing so far are a little uglier than method calls with regular arguments, what with all those curly braces:.

```
candidate = Candidate.new("Carl Barnes",
  {:age => 49, :occupation => "Attorney"})
```

...Which is why Ruby lets you leave the curly braces off, as long as the hash argument is the final argument:

```
candidate = Candidate.new("Carl Barnes",
        :age => 49, :occupation => "Attorney") No braces!
p candidate
#<Candidate:0x007fb412802c30
        @name="Carl Barnes", @age=49,
```

@name="Carl Barnes", @age=49, @occupation="Attorney", @hobby=nil, @birthplace=nil>

For this reason, you'll find that most methods that define a hash parameter define it as the last parameter.

Leave out the arrows!

Ruby offers one more shortcut we can make use of... If a hash uses symbols as keys, hash literals let you leave the colon (:) off the symbol and replace the hash rocket (=>) with a colon.

#<Candidate:0x007f9dc412aa98
@name="Amy Nguyen", @age=37,
@occupation="Engineer",
@hobby="Lacrosse",
@birthplace=nil>

Those hash arguments started out pretty ugly, we admit. But now that we know all the tricks to make them more readable, they're looking rather nice, don't you think? Almost like regular method arguments, but with handy labels next to them!

```
Candidate.new("Carl Barnes", age: 49, occupation: "Attorney")
Candidate.new("Amy Nguyen", age: 37, occupation: "Engineer")
```

there lare no Dumb Questions

Q: Is there anything special about a hash parameter? It looks like just another method parameter!

A: It is just another method parameter; there's nothing stopping you from passing an integer, a string, etc. when you should be passing a hash. But you're likely to get errors when your method code tries to access keys and values on an integer or string!



method that takes a hash parameter, ensure the hash parameter comes last, so that callers to your method can leave the curly braces off their hash. When calling a method with a hash argument, you should leave the curly braces off if possible - it's easier to read. And lastly, you should use symbols as keys whenever you're working with a hash parameter; it's more efficient.

Making the entire hash optional

There's one last improvement we can make to our Candidate class's initialize method. Currently we can include all of our hash keys:

```
Candidate.new("Amy Nguyen", age: 37, occupation: "Engineer",
hobby: "Lacrosse", birthplace: "Seattle")
```

Or we can leave *most* of them off:

```
Candidate.new("Amy Nguyen", age: 37)
```

But if we try to leave them *all* off, we get an error:

p Candidate.new("Amy Nguyen")



This happens because if we leave all the keys off, then as far as Ruby is concerned, we didn't pass a hash argument at all.

We can avoid this inconsistency by setting an empty hash as a default for the options argument:

Now, if no hash argument is passed, the empty hash will be used by default. All the Candidate attributes will be set to the nil default value from the empty hash.

p Candidate.new("Carl Barnes")

#<Candidate:0x007fbe0981ec18 @name="Carl Barnes", @age=nil, @occupation=nil, @hobby=nil, @birthplace=nil>

If we specify at least one key/value pair, though, the hash argument will be treated as before:

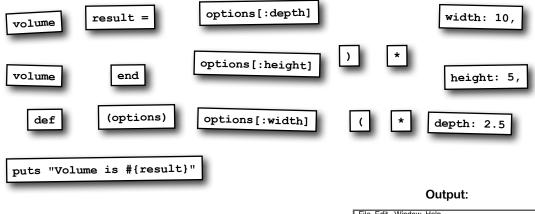
```
p Candidate.new("Carl Barnes", occupation: "Attorney")
```

```
#<Candidate:0x007fbe0981e970 @name="Carl Barnes", @age=nil,
@occupation="Attorney", @hobby=nil, @birthplace=nil>
```

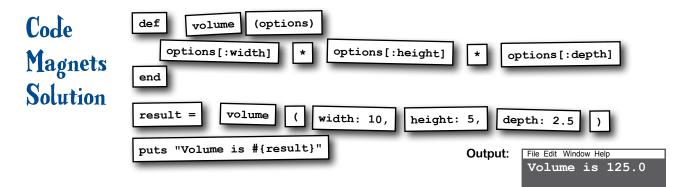


Code Magnets

A Ruby program is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working Ruby program that will produce the given output?



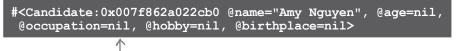
File Edit Wind	dow He	elp	
Volume	is	125.0	



Typos in hash arguments are dangerous

There's a downside to hash arguments that we haven't discussed yet, and it's just waiting to cause trouble for us... For example, you might expect this code to set the occupation attribute of the new Candidate instance, and you might be surprised when it doesn't:

p Candidate.new("Amy Nguyen", occupaiton: "Engineer")



- Why is this still nil?

Why didn't it work? Because we misspelled the symbol name in the hash key!

p Candidate.new("Amy Nguyen", occupaiton: "Engineer")

The code doesn't even raise an error. Our initialize method just uses the value of the correctly-spelled options [:occupation] key, which is of course nil, because it's never been assigned to.

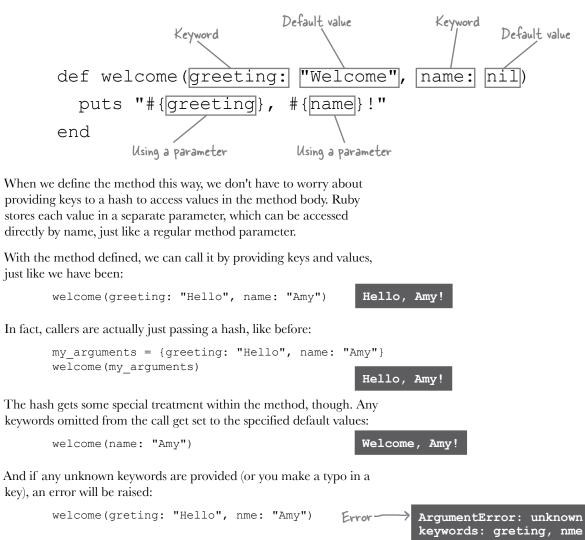
Silent failures now mean hard-to-diagnose bugs later. This doesn't make me want to use hash arguments...

0

Don't worry. In version 2.0, Ruby added <u>keyword</u> arguments, which can prevent this sort of issue.

Keyword arguments

Rather than require a single hash parameter in method definitions, we can specify the individual hash keys we want callers to provide, using this syntax:



Using keyword arguments with our Candidate class

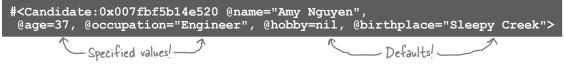
Currently, our Candidate class is using a hash parameter in its initialize method. The code is a bit ugly, and it won't warn a caller if they make a typo in a hash key.

Let's revise our Candidate class's initialize method to take keyword arguments.

We use "Sleepy Creek" as a default value for the birthplace keyword, and nil as a default for the others. We also replace all those references to the options hash in the method body with parameter names. The method is a lot easier to read now!

It can still be called the same way as before...

```
p Candidate.new("Amy Nguyen", age: 37, occupation: "Engineer")
```



...And it will warn callers if they make a typo in a keyword!

p Candidate.new("Amy Nguyen", occupaiton: "Engineer")

Required keyword arguments

Right now, we can still call Candidate.new even if we fail to provide the most basic information about a candidate..:

```
p Candidate.new("Carl Barnes")
```

All attributes are set to the defaults!

#<Candidate:0x007fe743885d38 @name="Carl Barnes", @age=nil, @occupation=nil, @hobby=nil, @birthplace="Sleepy Creek">

This isn't ideal. We want to require callers to provide at least an age and an occupation for a candidate.

Back when the initialize method was using ordinary method parameters, this wasn't a problem; *all* the arguments were required.

```
class Candidate
  attr_accessor :name, :age, :occupation, :hobby, :birthplace
  def initialize(name, age, occupation, hobby, birthplace)
    ...
  end
end
```

The only way to make a method parameter optional is to provide a default value for it.

```
class Candidate
  attr_accessor :name, :age, :occupation, :hobby, :birthplace
  def initialize(name, age = nil, occupation = nil, hobby = nil, birthplace = nil)
    ...
  end
end
```

But wait, we provide default values for all our keywords now ...

```
class Candidate
  attr_accessor :name, :age, :occupation, :hobby, :birthplace
  def initialize(name, age: nil, occupation: nil, hobby: nil, birthplace: "Sleepy Creek")
    ...
  end
end
```

If you take away the default value for an ordinary method parameter, that parameter is required; you can't call the method without providing a value. What happens if we take away the default values for our keyword arguments?

Required keyword arguments (cont.)

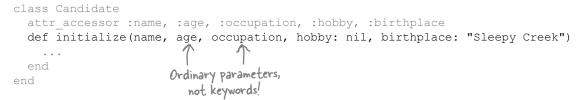
Let's try removing the default values for the age and occupation keywords, and see if they'll be required when calling initialize.



<u>Required</u> keyword arguments were only added in Ruby 2.1.

error if you're running Ruby 2.0, you'll get a syntax error if you try to use required keyword arguments. You'll need to either upgrade to 2.1, or provide default values.

We can't just remove the colon after the keyword, though. If we did, Ruby wouldn't be able to tell age and occupation apart from ordinary method parameters.



What if we removed the default value, but left the colon after the keyword?

```
class Candidate
  attr_accessor :name, :age, :occupation, :hobby, :birthplace
  def initialize(name, age:, occupation:, hobby: nil, birthplace: "Sleepy Creek")
    self.name = name
    self.age = age
    self.occupation = occupation
    self.hobby = hobby
    self.birthplace = birthplace
  end
end
```

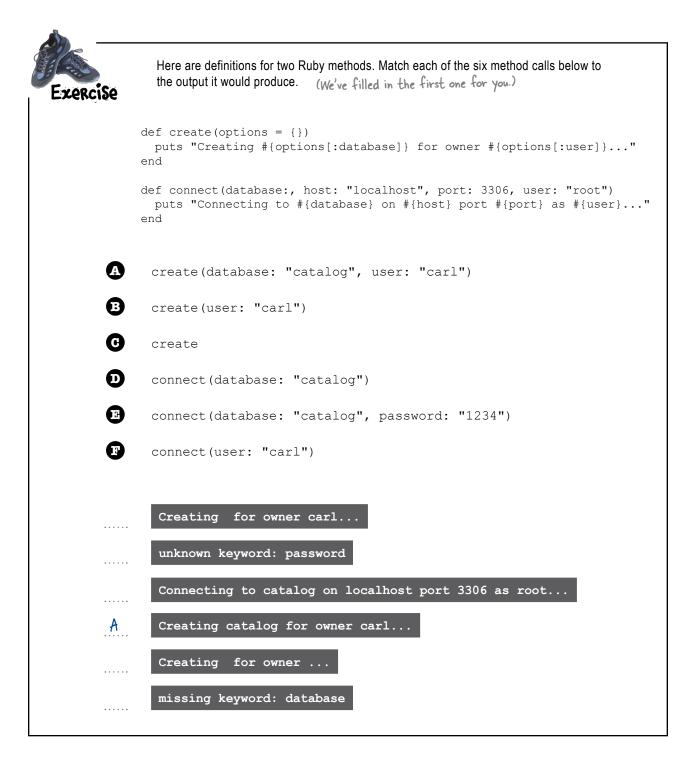
We can still call Candidate.new, as long as we provide the required keywords:

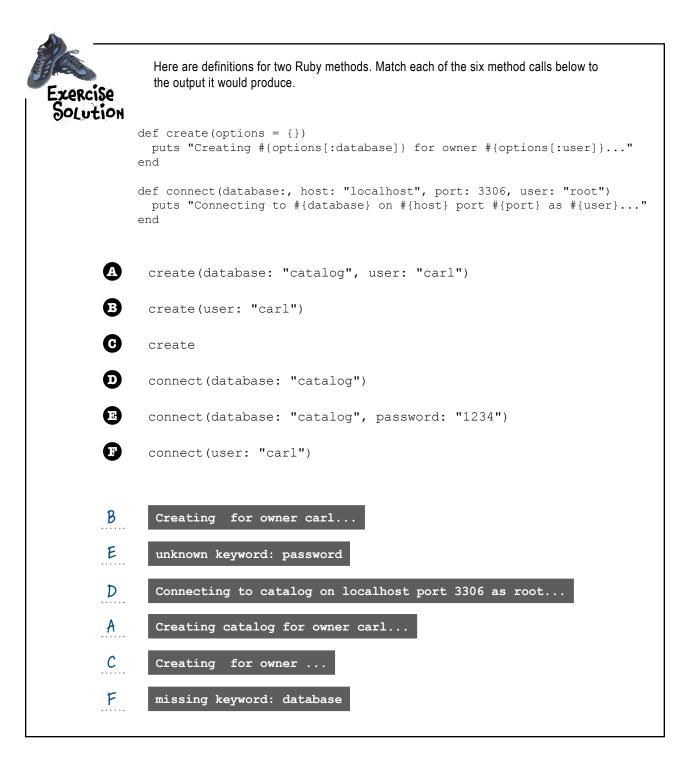
```
p Candidate.new("Carl Barnes", age: 49, occupation: "Attorney")
```

```
#<Candidate:0x007fcec281e5a0 @name="Carl Barnes",
@age=49, @occupation="Attorney", @hobby=nil, @birthplace="Sleepy Creek">
```

...And if we leave the required keywords off, Ruby will warn us!

You used to have to provide a long list of unlabelled arguments to Candidate.new, and you had to get the order *exactly* right. Now that you've learned to use hashes as arguments, whether explicitly or behind the scenes with keyword arguments, your code will be a lot cleaner!







Your Ruby Toolbox

That's it for Chapter 7! You've added hashes to your tool box.

Here are our notes on arrays from Chapter 5, - just for comparison.

Arrays

An array holds a collection of objects.

Arrays can be any size, and can grow or shrink as needed.

Arrays are ordinary Ruby objects, and have many useful instance

21-11-2 Hashes

A hash holds a collection of objects, each "labelled" with a key. You can use any object as a hash key. This is different than arrays, which can only use integers as

11-1

indexes. Hashes are also Ruby objects, and have many useful instance methods.

> .. And here are our notes for this chapter!



BULLET POINTS

- A hash literal is surrounded by curly braces. It needs to include a key for each value, like this: {"one" => 1, "two" => 2}
- When a hash key is accessed that a value has never been assigned to, nil is returned by default.
- Any Ruby expression can be used in conditional statements. Aside from the false boolean value, the only other value Ruby will treat as false is nil.
- You can use Hash, new instead of a hash literal to create a new hash. If you pass an object as an argument to Hash.new, that object will be returned by default when any key that hasn't been assigned to is accessed (instead of nil).
- If the key you access isn't exactly equal to the key in the hash, it will be treated as an entirely new key.
- Hashes have an each method that works a lot like the each method on arrays. The difference is that the block you provide should (normally) accept two parameters (instead of one): one for each key, and one for the corresponding value.
- If you pass a hash as the last argument to a method, Ruby lets you leave the braces off.
- If a hash uses symbols as keys, you can leave the colon off the symbol, and replace => with a colon, like this:

{name: "Kim", age: 28}

- When defining a function, you can specify that callers should provide keyword arguments. The keywords and values are actually just a hash behind the scenes, but the values are placed into named parameters within the function.
- Keyword arguments can be required, or they can be made optional by defining a default value.

8 references

Crossed Signals



Ever sent an e-mail to the wrong contact? You probably had a hard time sorting out the confusion that ensued. Well, *Ruby objects* are *just like those contacts* in your address book, and *calling methods* on them is like *sending messages* to them. If your address book gets *mixed up*, it's possible to send the wrong message to the wrong object. And you'll have a hard time sorting out the confusion that ensues. This chapter will help you *recognize the signs* that messages are going to the wrong objects, and help you *get your programs running smoothly* again.

Some confusing bugs

The word continues to spread - if someone has a Ruby problem, your company can solve it. And so people are showing up at your door with some unusual problems...

This astronomer thinks he has a clever way to save some coding... Instead of typing my_star = CelestialBody.new and my_star.type = 'star' for every star he wants to create, he wants to just *copy* the original star, and set a new name for it.

```
class CelestialBody
  attr accessor :type, :name
end
altair = CelestialBody.new
altair.name = 'Altair' To save time, he wants to
                          _ copy the previous star...
altair.type = 'star'
polaris = altair <del><</del>
                                  -...And just change
polaris.name = 'Polaris' <
                                   the name.
vega = polaris
                           -Same
vega.name = 'Vega' <
                            here.
puts altair.name, polaris.name, vega.name
                                                      Vega
                                                                    But it looks like the
                                                      Vega
                                                      Vega
                                                                    names on all 3 stars
```

0

are now identical!

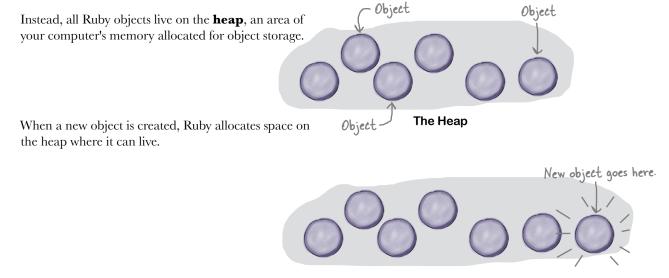
But the plan seems to be backfiring. All three of his CelestialBody instances are reporting that they have the *same* name!

The heap

The bug in the star catalog program stems from an underlying problem: the developer *thinks* he's working with *multiple* objects, when actually he's operating on the *same* object over and over.

To understand how that can be, we're going to need to learn about where objects really live, and how your programs communicate with them.

Rubyists often talk about "placing objects in variables", "storing objects in arrays", "storing an object in a hash value", and so forth. But that's just a simplification of what actually happens. Because you can't actually put an object *in* a variable, array, or hash.



Generally, you don't need to concern yourself with the heap - Ruby manages it for you. The heap grows in size if more space is needed. Objects that are no longer used get cleared off the heap. It's not something you usually have to worry about.

But we *do* need to be able to *retrieve* items that are stored on the heap. And we do that with *references*. Read on to learn more about them.

References

When you want to send a letter to a particular person, how do you get it to them? Each residence in a city has an *address* that mail can be sent to. You simply write the address on an envelope. A postal worker then uses that address to find the residence and deliver the letter.

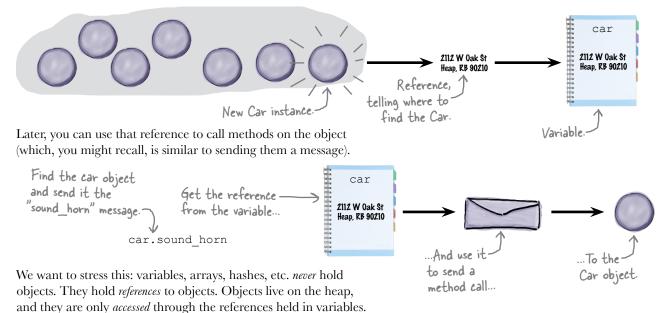


When a friend of yours moves into a new residence, they give you their address, which you then write down in an address book or other convenient place. This allows you to communicate with them in the future.



Similar to addresses for houses, Ruby uses **references** to locate objects on the heap. When a new object is created, it returns a reference to itself. You store that reference in a variable, array, or other convenient place. Kind of like a house address, the reference tells Ruby where the object "lives" on the heap.

Reference stored here. ____ car = car.new ____ Returns a reference to the new Car.



Candace Camden

2110 W Oak St Heap, RB 90210

When references go wrong

Andy met not one, but *two* gorgeous women last week: Betty and Candace. Better yet, they both live on his street.

> Andy Adams 2100 W Oak St —— Heap, RB 90210

Andy intended to write both their addresses down in his address book. Unfortunately for him, he accidentally wrote the *same* address (Betty's) down for *both* women. **Oak Street in reality**

Betty Bell

2106 W Oak St

Heap, RB 90210



Oak Street according to Andy's address book

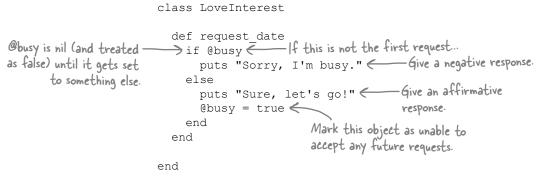
Later that week, Betty received <i>two</i> letters from Andy:	Dear Betty. It was great to meet you on Tuesday! I really enjoyed chatting about that thing you like. Say, I was wondering, would you go to the big dance with me next week? I think we'd have a good time. Let me know! Yours, Andy	Pear Candace, It was great to meet you on Monday! I really enjoyed chatting about that thing you like. Say, I was wondering, would you go to the movies with me next week? I think we'd have a good time. Let me know! Yours, Andy
---	---	---

Now, Betty is angry at Andy, and Candace (who never received a letter) thinks Andy is ignoring her.

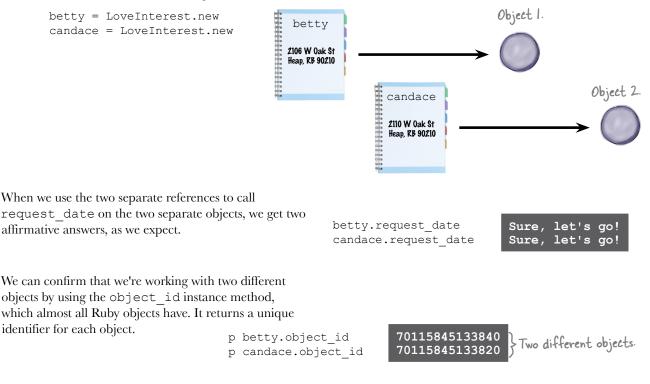
What does any of this have to do with fixing our Ruby programs? You're about to find out...

Aliasing

Andy's dilemma can be simulated in Ruby with this simple class, called LoveInterest. A LoveInterest has an instance method, request_date, which will print an affirmative response just once. If the method is called again after that, the LoveInterest will report that it's busy.

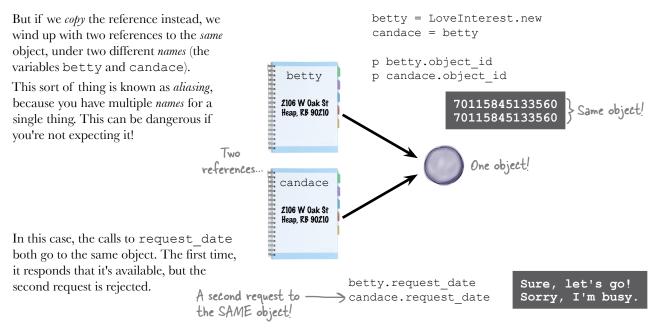


Normally, when using this class, we would create two separate objects, and store references to them in two separate variables:



Aliasing (cont.)

.....



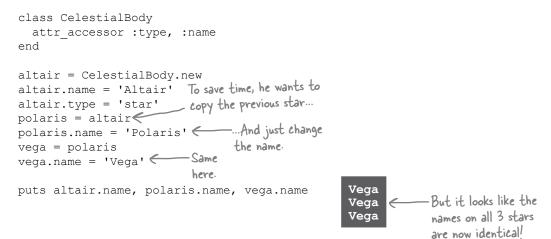
This aliasing behavior seems *awfully* familiar... Remember the malfunctioning star catalog program? Let's go back and take another look at that next.

Exercise	Here is a Ruby class:	<pre>class Counter def initialize @count = 0 end def increment @count += 1 puts @count end</pre>	And here is some code that uses that class:	<pre>a = Counter.r b = Counter.r c = b d = c a.increment b.increment c.increment d.increment</pre>	-
		end	Guess what the code write your answer in t (We've filled in the f	he blanks.	1

Exercise Solution	Here is a Ruby class:	<pre>class Counter def initialize @count = 0 end def increment @count += 1 puts @count end</pre>	And here is some Ruby code that uses that class:	<pre>a = Counter.: b = Counter.: c = b d = c a.increment b.increment d.increment</pre>	-
		end	Guess what the code w write your answer in th		 2 3

Fixing the astronomer's program

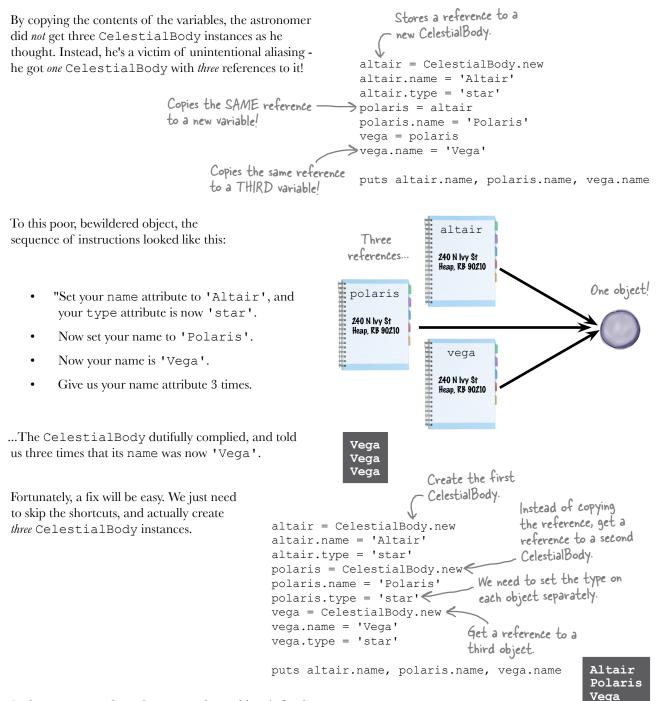
Now that we've learned about aliasing, let's take another look at the astronomer's malfunctioning star catalog, and see if we can figure out the problem this time...



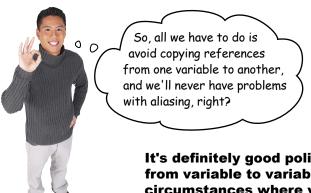
If we try calling object_id on the objects in the three variables, we'll see that all three variables refer to the *same* object. The same object under three different names... sounds like another case of aliasing!

```
puts altair.object_id
puts polaris.object_id
puts vega.object_id
To189936850940
Same object!
Same object!
```

Fixing the astronomer's program (cont.)



And as we can see from the output, the problem is fixed!



It's definitely good policy to avoid copying references from variable to variable. But there are other circumstances where you need to be aware of how aliasing works, as we'll see shortly.

Quickly identifying objects with "inspect"

Before we move on, we should mention a shortcut for identifying objects... We've already shown you how to use the object_id instance method. If it outputs the same value for the object in two variables, you know they both point to the same object.

```
altair = CelestialBody.new
altair.name = 'Altair' Copies the SAME reference
altair.type = 'star' to a new variable!
polaris = altair
polaris.name = 'Polaris'
puts altair.object_id, polaris.object_id
7035
```

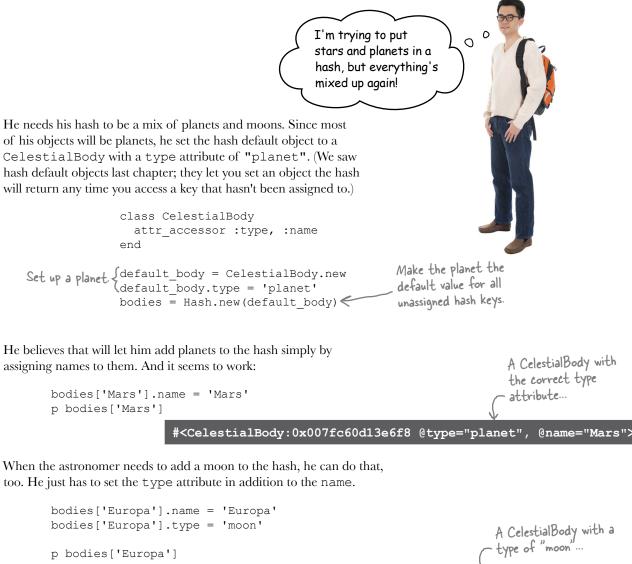


The string returned by the inspect instance method also includes a representation of the object ID, in hexadecimal (consisting of the numbers "0" through "9" and the letters "a" through "f"). You don't need to know the details of how hexadecimal works; just know that if you see the *same value* for the object referenced by two variables, you have two aliases for the *same object*. A *different value* means a *different object*.

```
puts altair.inspect, polaris.inspect
vega = CelestialObject.new
puts vega.inspect
The SAME object! {
    #<CelestialBody:0x007ff76b17f100 @name="Polaris", @type="star">
    #<CelestialBody:0x007ff76b17edb8>
```

Problems with a hash default object

The astronomer is back, with more problematic code...



p bodies['Europa']

#<CelestialBody:0x007fc60d13e6f8 @type="moon", @name="Europa">

But then, as he continues adding new CelestialBody objects to the hash, it starts behaving strangely ...

Problems with a hash default object (cont.)

The problems with using a CelestialBody as a hash default object become apparent as more objects as the astronomer tries to add more objects to the hash... When he adds another planet after adding a moon, the planet's type attribute is set to "moon" as well!

```
This is supposed to be a planet.
- Why is this set to "moon"?!
        bodies['Venus'].name = 'Venus'
        p bodies['Venus']
                           #<CelestialBody:0x007fc60d13e6f8 @type="moon", @name="Venus">
... If he goes back and gets the value for the keys he added previously,
                                                                                    What happened to
those objects appear to have been modified as well!
                                                        Isn't one of these supposed to be a "moon"?-
                                                                                    the names "Mars'
        p bodies['Mars']
                                                                                       and "Europa"?
        p bodies['Europa']
                           #<CelestialBody:0x007fc60d13e6f8 @type="moon", @name="Venus">
                           #<CelestialBody:0x007fc60d13e6f8 @type="moon", @name="Venus">
                                     But we're not altering multiple
                                     objects... Look at the object
                                                                            C
                                     IDs. All these different hash
                                     keys are giving us references
                                     to the same object!
Good observation! Remember we said that the inspect method
string includes a representation of the object ID? And as you know,
the p method calls inspect on each object before printing it. Using
the p method shows us that all the hash keys refer to the same object!
        p bodies['Venus']
                                            These are all the
        p bodies['Mars']
                                              SAME object !-
        p bodies['Europa']
                              #<CelestialBody:0x007fc60d13e6f8 @type="moon",</pre>
                                                                                          @name="Venus">
                              #<CelestialBody:0x007fc60d13e6f8 @type="moon", @name="Venus">
                              #<CelestialBody:0x007fc60d13e6f8 @type="moon",</pre>
                                                                                          @name="Venus">
```

Looks like we've got a problem with aliasing again! On the next few pages, we'll see how to fix it.

We're actually modifying the hash default object!

The central problem with this code is that we're not actually modifying hash values. Instead, we're modifying the *hash default object*.

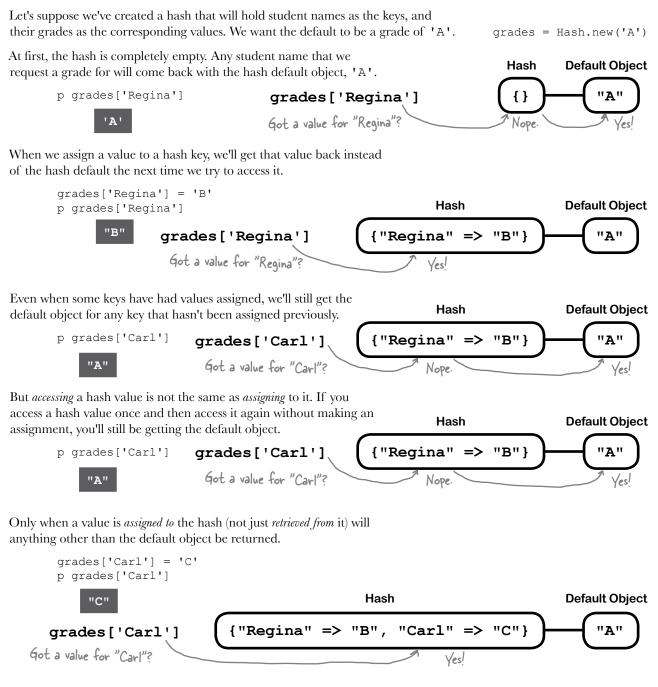
We can confirm this using the default instance method, which is available on all hashes. It lets us look at the default object after we create the hash.

Let's inspect the default object both before and after we attempt to add a planet to the hash.	<pre>class CelestialBody attr_accessor :type, :name end</pre>
	default_body = CelestialBody.new default_body.type = 'planet' bodies = Hash.new(default_body)
	p bodies.default
	bodies['Mars'].name = 'Mars'
	p bodies.default
	CelestialBody:0x007f868a8274c8 @type="planet"> CelestialBody:0x007f868a8274c8 @type="planet", @name="Mars">
	The name got added to the default object instead!
So why is a name being added to the getting added to the hash value for	ne default object? Shouldn't it be
If we look at the object IDs for both bodies['Mars'] and	
the hash default object, we'll have our answer:	p bodies['Mars'] p bodies.default Same object ID!
	CelestialBody:0x007f868a8274c8 @type="planet", @name="Mars"> CelestialBody:0x007f868a8274c8 @type="planet", @name="Mars">

When we access bodies ['Mars'], we're still getting a reference to the hash default object! But why?

A more detailed look at hash default objects

When we introduced the hash default object in the last chapter, we said that you get the default object anytime you access a key *that hasn't been assigned to yet*. Let's take a closer look at that last detail.



class CelestialBody

end

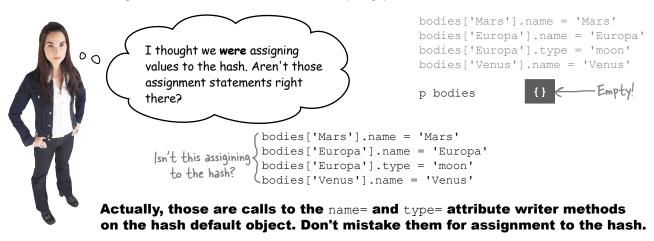
attr accessor :type, :name

default body = CelestialBody.new

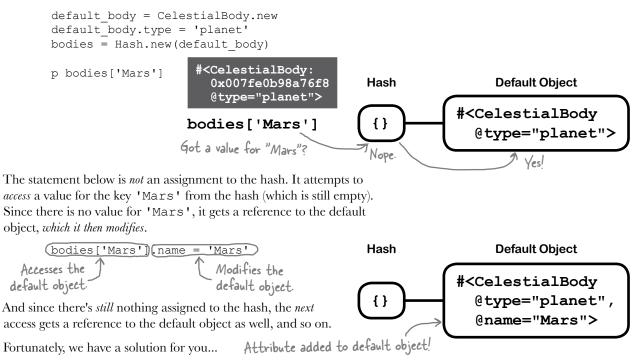
default_body.type = 'planet'
bodies = Hash.new(default body)

Back to the hash of planets and moons

And *that* is why, when we try to set the type and name attributes of objects in the hash of planets and moons, we wind up altering the default object instead. We're not actually assigning any values to the hash. In fact, if we inspect the hash itself, we'll see that it's totally empty!



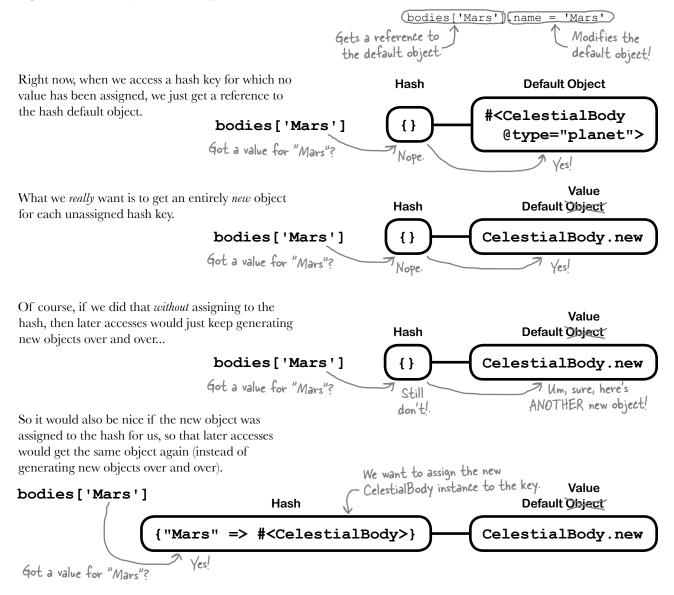
When we access a key for which no value has been assigned, we get the default object back.



Our wish list for hash defaults

We've determined that this code doesn't *assign* a value to the hash, it just *accesses* a value. It gets a reference to the default object, which it then (unintentionally) modifies.

```
default_body = CelestialBody.new
default_body.type = 'planet'
bodies = Hash.new(default body)
```

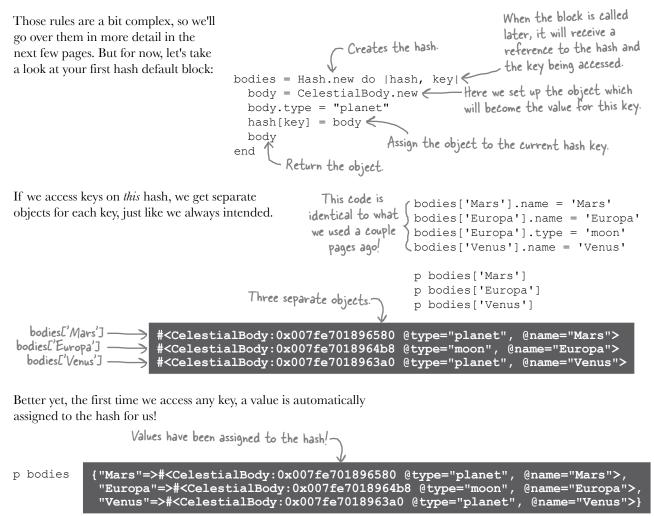


Hashes have a feature that can do all this for us!

Hash default blocks

Instead of passing an argument to Hash.new to be used as a hash default *object*, you can pass a *block* to Hash.new to be used as the hash default *block*. When a key is accessed for which no value has been assigned:

- The block is called.
- The block receives references to the hash and the current key as block parameters. These can be used to assign a value to the hash.
- The block return value is returned as the current value of the hash key.



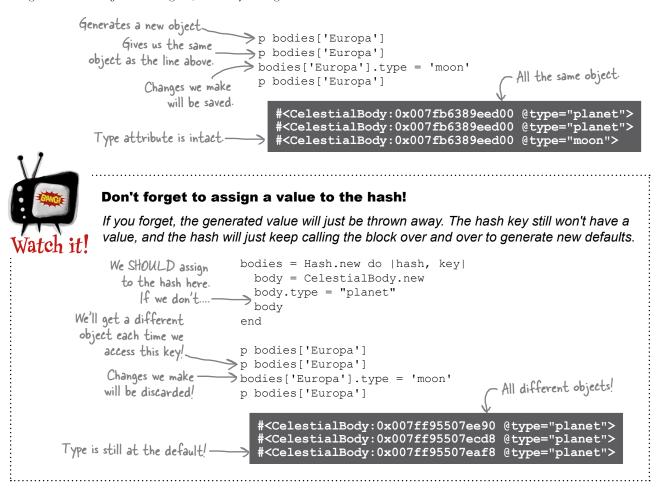
Now that we know it will work, let's take a closer look at the components of that block...

Hash default blocks: Assigning to the hash

In most cases, you'll want the value created by your hash default block to be assigned to the hash. A reference to the hash and the current key are passed to the block, in order to allow you to do so. When the block is called later, it will receive a reference to the hash and the key being accessed.

bodies = Hash.new do |hash, key|
body = CelestialBody.new
body.type = "planet"
hash[key] = body
body
end
Assign the object to the current hash key.

When we assign values to the hash in the block body, things work like we've been expecting all along. A new object is generated for each new key you access. On subsequent accesses, we get the same object back again, with any changes we've made intact.



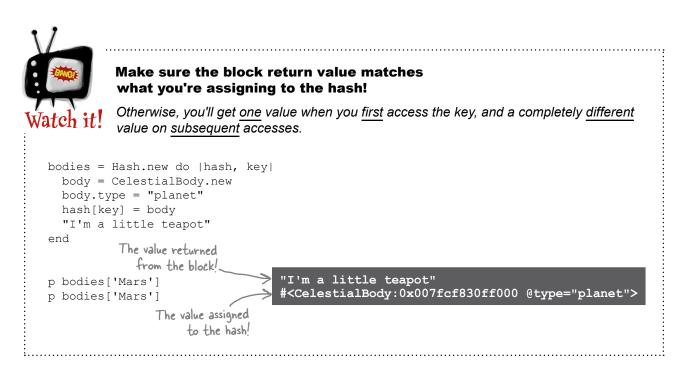
Hash default blocks: Block return value

When you access an unassigned hash key for the first time, the hash default block's return value is returned as the value for the key.

```
bodies = Hash.new do |hash, key|
body = CelestialBody.new
body.type = "planet"
hash[key] = body
body This return value...
end
p bodies['Mars'] [s what we get here!
```

#<CelestialBody:0x007fef7a9132c0 @type="planet">

As long as you assign a value to the key within the block body, the hash default block won't be invoked for subsequent accesses of that key; instead, you'll get whatever value was assigned.



Generally speaking, you won't need to work very hard to remember this rule. As we'll see on the next page, setting up an appropriate return value for your hash default block happens quite naturally...

Hash default blocks: A shortcut

Thus far, we've been returning a value from the hash default block on a separate line:

p bodies['Mars']

 $p my hash = \{\}$

p my array = []

p my integer = 20

p my array[0] = 245

p my hash['A'] = ['Apple']

#<CelestialBody:0x007fef7a9132c0 @type="planet">

{ }

[]

20

245

["Apple"]

Values of expressions

same as values assigned

But Ruby offers a shortcut that can reduce the amount of code in your default block a bit...

You've already learned that the value of the last expression in a block is treated as the block's return value... What we *haven't* mentioned is that in Ruby, the value of an assignment expression is the same as the value being assigned.

So, we can use just an assignment statement in a hash default block, and it will return the assigned value.

```
greetings = Hash.new do |hash, key|
hash[key] = "Hi, #{key}"
end
p greetings["Kayla"] "Hi, Kayla"
```

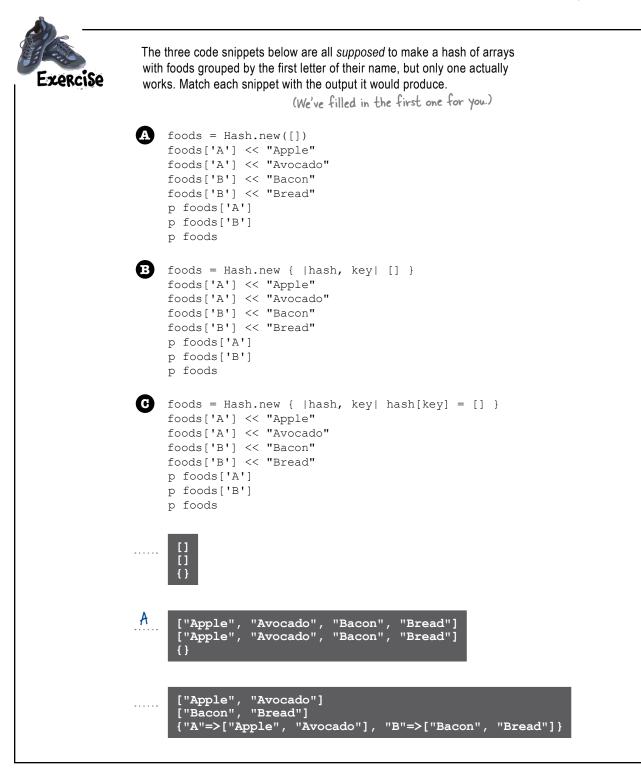
And, of course, it will add the value to the hash as well.

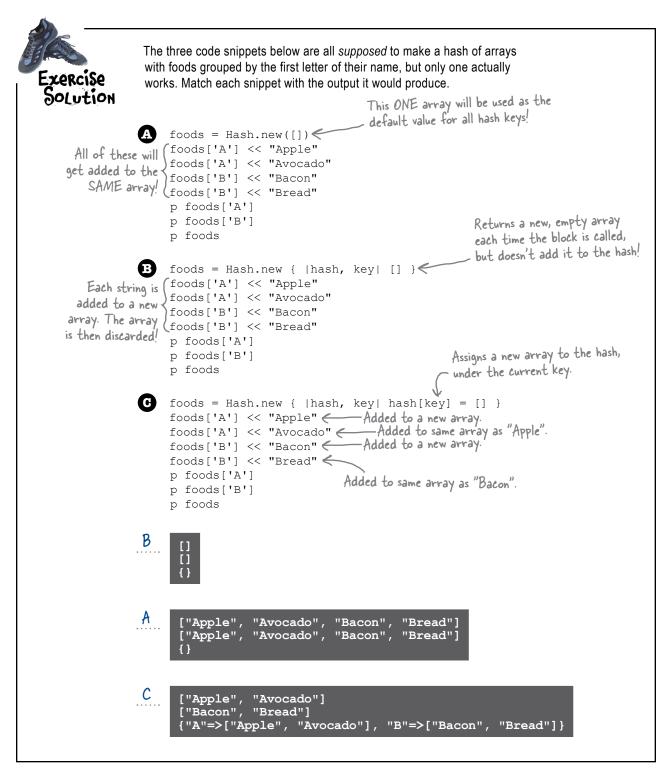
```
p greetings {"Kayla"=>"Hi, Kayla"}
```

So in the astronomer's hash, instead of adding a separate line with a return value, we can just let the value of the assignment expression provide the return value for the block.

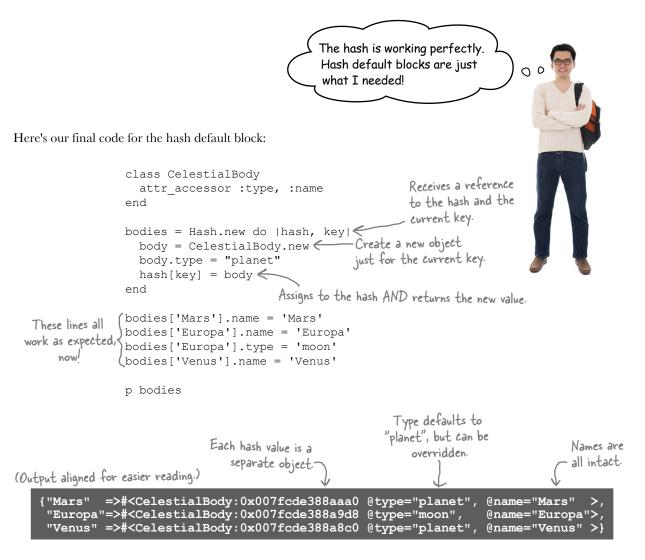
```
p bodies['Mars']
```







The astronomer's hash: our final code



Here's what we did to get this program working:

- We use a hash default block to create a *unique* object for each hash key. (This is unlike a hash default object, which gives references to *one* object as the default for *all* keys.)
- Within the block, we assign the new object to the current hash key.
- The new object becomes the value of the assignment expression, which also becomes the block's return value. So the first time a given hash key is accessed, they get the new object as the corresponding value.

00

Using hash default objects safely

I have one more question. Why would anyone use a hash default object when you can use a hash default block instead?

Hash default objects work very well if you use a number as the default.

I should only use numbers? Then why did Ruby let us use a CelestialBody as a default object earlier, without even a warning?

0

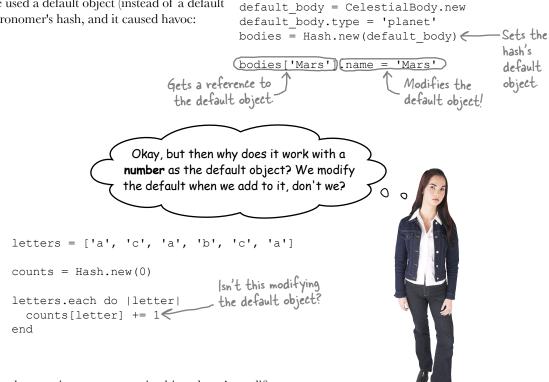
Okay, it's a little more complicated than that. Hash default objects work very well <u>if</u> you don't change the default, and <u>if</u> you assign values back to the hash. It's just that numbers make it easy to follow these rules.

Take this example, which counts the number of times letters occur in an array. (It works just like the vote counting code from last chapter.)

Using a hash default object here works because we follow the above two rules...

Hash default object rule #1: Don't modify the default object

If you're going to use a hash default object, it's important not to modify that object. Otherwise, you'll get unexpected results the next time you access the default. We saw this happen when we used a default object (instead of a default block) for the astronomer's hash, and it caused havoc:



In Ruby, doing math operations on a numeric object doesn't modify that object; it returns an entirely *new* object. We can see this if we look at object IDs before and after an operation.

```
number = 0
puts number.object_id
number = number + 1
puts number.object_id
```

 Two different objects! (Object IDs for integers are much lower than for
 other objects, but that's an implementation detail, so don't worry about it. The key point is, they're different.)

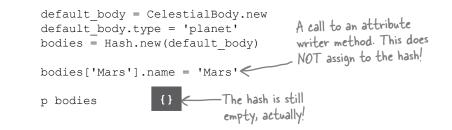
In fact, numeric objects are *immutable*: they don't *have* any methods that modify the object's state. Any operation that might change the number gives you back an entirely new object.

That's what makes numbers safe to use as hash default objects; you can be certain that the default number won't be changed accidentally.

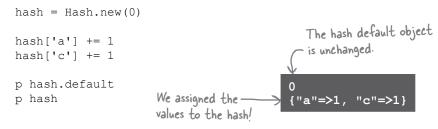
Numbers make good hash default objects because they are <u>immutable</u>.

Hash default object rule #2: Assign values to the hash

If you're going to use a hash default object, it's also important to ensure you're actually assigning values to the hash. As we saw with the astronomer's hash, sometimes it can look like you're assigning to the hash when you're not...



When we use a *number* as a default object, though, it's much more natural to actually assign values to the hash. (Because numbers are immutable, we *can't* store the incremented values *unless* we assign them to the hash!)



The rule of thumb for hash defaults



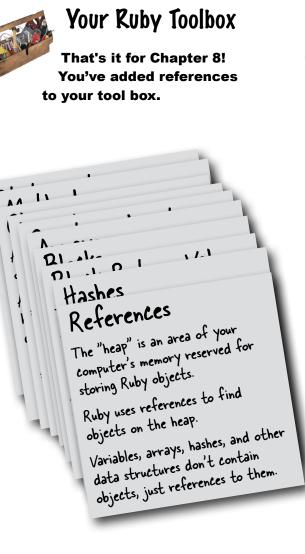
All of this seems like a lot to remember, just to be able to use hash defaults.

That's true. So we have a rule of thumb that will keep you out of trouble...

If your default is a <u>number</u>, you can use a hash default <u>object</u>. If your default is <u>anything else</u>, you should use a hash default <u>block</u>.

As you gain more experience with references, all of this will become second nature, and you can break this rule of thumb when the time is right. Until then, this should prevent most problems you'll encounter.

Understanding Ruby references and the issue of aliasing won't help you write more powerful Ruby programs. It *will* help you quickly find and fix problems when they arise, however. Hopefully this chapter has helped you form a basic understanding of how references work, and will let you avoid trouble in the first place.



BULLET POINTS

- If you need to store more objects, Ruby will increase the size of the heap for you. If you're no longer using objects, Ruby will delete them from the heap for you.
- Aliasing is the copying of a reference to an object, and it can cause bugs if you do it unintentionally.
- Most Ruby objects have an object_id instance method, which returns a unique identifier for the object. It can be used to determine whether you have multiple references to a single object.
- The string returned by the inspect method also includes a representation of the object ID.
- If you set a default object for a hash, all unassigned hash keys will return references to that single default object.
- For this reason, it's best to only use immutable objects (objects that can't be modified), such as numbers, as hash default objects.
- If you need any other kind of object as a hash default, it's better to use a hash default block, so that a unique object is created for each key.
- Hash default blocks receive a reference to the hash and the current key as block parameters. In most cases, you'll want to use these parameters to assign a new object as a value for the given hash key.
- The hash default block's return value is treated as the initial default value for the given key.
- The value of a Ruby assignment expression is the same as the value being assigned. So if an assignment expression is the last expression in a block, the value assigned becomes the block's return value.