Chapter 2

Ruby.new

Most books on programming languages look about the same. They start with chapters on basic types: integers, strings, and so on. Then they look at expressions, before moving on to if and while statements. Then, perhaps around Chapter 7 or 8, they'll start mentioning classes. We find that somewhat tedious.

Instead, when we designed this book, we had a grand plan (we were younger then). We wanted to document the language from the top down, starting with classes and objects and ending with the nitty-gritty syntax details. It seemed like a good idea at the time. After all, most everything in Ruby is an object, so it made sense to talk about objects first.

Or so we thought.

Unfortunately, it turns out to be difficult to describe a language that way. If you haven't covered strings, if statements, assignments, and other details, it's difficult to write examples of classes. Throughout our top-down description, we kept coming across low-level details we needed to cover so that the example code would make sense.

So, we came up with another grand plan (they don't call us pragmatic for nothing). We'd still describe Ruby starting at the top. But before we did that, we'd add a short chapter that described all the common language features used in the examples along with the special vocabulary used in Ruby, a kind of mini-tutorial to bootstrap us into the rest of the book. And that mini-tutorial is this chapter.

Ruby Is an Object-Oriented Language

Let's say it again. Ruby is a genuine object-oriented language. Everything you manipulate is an object, and the results of those manipulations are themselves objects. However, many languages make the same claim, and their users often have a different interpretation of what *object-oriented* means and a different terminology for the concepts they employ.

So, before we get too far into the details, let's briefly look at the terms and notation that *we'll* be using.

When you write object-oriented programs, you're normally looking to model concepts from the real world. Typically during this modeling process you'll discover categories of things that need to be represented in code. In a jukebox, the concept of a "song" could be such a category. In Ruby, you'd define a *class* to represent each of these entities. A class is a combination of state (for example, the name of the song) and methods that use that state (perhaps a method to play the song).

Once you have these classes, you'll typically want to create a number of *instances* of each. For the jukebox system containing a class called Song, you'd have separate instances for popular hits such as "Ruby Tuesday," "Enveloped in Python," "String of Pearls," "Small Talk," and so on. The word *object* is used interchangeably with *class instance* (and being lazy typists, we'll probably be using the word *object* more frequently).

In Ruby, these objects are created by calling a *constructor,* a special method associated with a class. The standard constructor is called new.

Download **samples/intro_1.rb**

```
song1 = Song.new("Ruby Tuesday")
song2 = Song.new("Enveloped in Python")
# and so on
```

These instances are both derived from the same class, but they have unique characteristics. First, every object has a unique *object identifier* (abbreviated as *object ID*). Second, you can define *instance variables*, variables with values that are unique to each instance. These instance variables hold an object's state. Each of our songs, for example, will probably have an instance variable that holds the song title.

Within each class, you can define *instance methods*. Each method is a chunk of functionality that may be called in the context of the class and (depending on accessibility constraints) from outside the class. These instance methods in turn have access to the object's instance variables and hence to the object's state. A Song class, for example, might define an instance method called play. If the variable my_way referenced a particular Song instance, you'd be able to call that instance's play method and play a particular song.

Methods are invoked by sending a message to an object. The message contains the method's name, along with any parameters the method may need.[1] When an object receives a message, it looks into its own class for a corresponding method. If found, that method is executed. If the method *isn't* found. . . well, we'll get to that later.

This business of methods and messages may sound complicated, but in practice it is very natural. Let's look at some method calls. In this code, we're using puts, a standard Ruby method that writes its argument(s) to the console, adding a newline after each:

```
puts "gin joint".length
puts "Rick".index("c")
puts 42.even?
puts sam.play(song)
```

1. This idea of expressing method calls in the form of messages comes from Smalltalk.

produces:

```
9
2
true
duh dum, da dum de dum ...
```

Each line shows a method being called as an argument to puts. The thing before the period is called the *receiver*, and the name after the period is the method to be invoked. The first example asks a string for its length, and the second asks a different string to find the index of the letter *c*. The third line asks the number 42 if it is even (the question mark is part of the method name even?). Finally, we ask Sam to play us a song (assuming there's an existing variable called sam that references an appropriate object).

It's worth noting here a major difference between Ruby and most other languages. In (say) Java, you'd find the absolute value of some number by calling a separate function and passing in that number. You could write this:

```
num = Math.abs(num)     // Java code
```

In Ruby, the ability to determine an absolute value is built into numbers—they take care of the details internally. You simply send the message abs to a number object and let it do the work:

```
num      = -1234     # =>   -1234
positive = num.abs   # =>   1234
```

The same applies to all Ruby objects. In C you'd write strlen(name), but in Ruby it's name.length, and so on. This is part of what we mean when we say that Ruby is a genuine object-oriented language.

Some Basic Ruby

Not many people like to read heaps of boring syntax rules when they're picking up a new language, so we're going to cheat. In this section, we'll hit some of the highlights—the stuff you'll just *need* to know if you're going to write Ruby programs. Later, in Chapter 22, which begins on page 325, we'll go into all the gory details.

Let's start with a simple Ruby program. We'll write a method that returns a cheery, personalized greeting. We'll then invoke that method a couple of times:

Download **samples/intro_5.rb**

```
def say_goodnight(name)
  result = "Good night, " + name
  return result
end
# Time for bed...
puts say_goodnight("John-Boy")
puts say_goodnight("Mary-Ellen")
```

As the example shows, Ruby syntax is clean. You don't need semicolons at the ends of statements as long as you put each statement on a separate line. Ruby comments start with a # character and run to the end of the line. Code layout is pretty much up to you; indentation is not significant (but using two-character indentation will make you friends in the community if you plan on distributing your code).

Methods are defined with the keyword def, followed by the method name (in this case, say_goodnight) and the method's parameters between parentheses. (In fact, the parentheses are optional, but we like to use them.) Ruby doesn't use braces to delimit the bodies of compound statements and definitions. Instead, you simply finish the body with the keyword end. Our method's body is pretty simple. The first line concatenates the literal string "Good night,␣" and the parameter name and assigns the result to the local variable result. The next line returns that result to the caller. Note that we didn't have to declare the variable result; it sprang into existence when we assigned to it.

Having defined the method, we invoke it twice. In both cases, we pass the result to the method puts, which simply outputs its argument followed by a newline (moving on to the next line of output):

```
Good night, John-Boy
Good night, Mary-Ellen
```

The line

```
puts say_goodnight("John-Boy")
```

contains two method calls, one to the method say_goodnight and the other to the method puts. Why does one call have its arguments in parentheses while the other doesn't? In this case, it's purely a matter of taste. The following lines are both equivalent:

```
puts say_goodnight("John-Boy")
puts(say_goodnight("John-Boy"))
```

However, life isn't always that simple, and precedence rules can make it difficult to know which argument goes with which method invocation, so we recommend using parentheses in all but the simplest cases.

This example also shows some Ruby string objects. Ruby has many ways to create a string object, but probably the most common is to use *string literals*, which are sequences of characters between single or double quotation marks. The difference between the two forms is the amount of processing Ruby does on the string while constructing the literal. In the single-quoted case, Ruby does very little. With a few exceptions, what you type into the string literal becomes the string's value.

In the double-quoted case, Ruby does more work. First, it looks for substitutions (sequences that start with a backslash character) and replaces them with some binary value. The most common of these is \n, which is replaced with a newline character. When a string containing a newline is output, that newline becomes a line break:

```
puts "And good night,\nGrandma"
```

produces:

```
And good night,
Grandma
```

The second thing that Ruby does with double-quoted strings is expression interpolation. Within the string, the sequence #{*expression*} is replaced by the value of *expression*. We could use this to rewrite our previous method:

Download **samples/intro_10.rb**

```
def say_goodnight(name)
  result = "Good night, #{name}"
  return result
end
puts say_goodnight('Pa')
```

produces:

```
Good night, Pa
```

When Ruby constructs this string object, it looks at the current value of name and substitutes it into the string. Arbitrarily complex expressions are allowed in the #{...} construct. In the following example, we invoke the capitalize method, defined for all strings, to output our parameter with a leading uppercase letter:

Download **samples/intro_11.rb**

```
def say_goodnight(name)
  result = "Good night, #{name.capitalize}"
  return result
end
puts say_goodnight('uncle')
```

produces:

```
Good night, Uncle
```

For more information on strings, as well as on the other Ruby standard types, see Chapter 6, which begins on page 106.

Finally, we could simplify this method some more. The value returned by a Ruby method is the value of the last expression evaluated, so we can get rid of the temporary variable and the return statement altogether:

Download **samples/intro_12.rb**

```
def say_goodnight(name)
  "Good night, #{name.capitalize}"
end
puts say_goodnight('ma')
```

produces:

```
Good night, Ma
```

We promised that this section would be brief. We have just one more topic to cover: Ruby names. For brevity, we'll be using some terms (such as *class variable*) that we aren't going to define here. However, by talking about the rules now, you'll be ahead of the game when we actually come to discuss class variables and the like later.

Ruby uses a convention that may seem strange at first: the first characters of a name indicate how the name is used. Local variables, method parameters, and method names should all

start with a lowercase letter or with an underscore. Global variables are prefixed with a dollar sign ($), and instance variables begin with an "at" sign (@). Class variables start with two "at" signs (@@).[2] Finally, class names, module names, and constants must start with an uppercase letter. Samples of different names are given in Table 2.1 on the next page.

Following this initial character, a name can be any combination of letters, digits, and underscores (with the proviso that the character following an @ sign may not be a digit). However, by convention, multiword instance variables are written with underscores between the words, and multiword class names are written in MixedCase (with each word capitalized). Method names may end with the characters ?, !, and =.

Arrays and Hashes

Ruby's arrays and hashes are indexed collections. Both store collections of objects, accessible using a key. With arrays, the key is an integer, whereas hashes support any object as a key. Both arrays and hashes grow as needed to hold new elements. It's more efficient to access array elements, but hashes provide more flexibility. Any particular array or hash can hold objects of differing types; you can have an array containing an integer, a string, and a floating-point number, as we'll see in a minute.

You can create and initialize a new array object using an *array literal*—a set of elements between square brackets. Given an array object, you can access individual elements by supplying an index between square brackets, as the next example shows. Note that Ruby array indices start at zero.

Download **samples/intro_13.rb**

```
a = [ 1, 'cat', 3.14 ]   # array with three elements
puts "The first element is #{a[0]}"
# set the third element
a[2] = nil
puts "The array is now #{a.inspect}"
```

produces:

```
The first element is 1
The array is now [1, "cat", nil]
```

You may have noticed that we used the special value nil in this example. In many languages, the concept of *nil* (or *null*) means "no object." In Ruby, that's not the case; nil is an object, just like any other, that happens to represent nothing. Anyway, let's get back to arrays and hashes.

2. Although we talk about global and class variables here for completeness, you'll find they are rarely used in Ruby programs. There's a lot of evidence that global variables make programs harder to maintain. Class variables are not as dangerous—it's just that people tend not to use them much.

Table 2.1. Example Variable and Class Names

Variables				Constants and
Local	**Global**	**Instance**	**Class**	**Class Names**
name	$debug	@name	@@total	PI
fish_and_chips	$CUSTOMER	@point_1	@@symtab	FeetPerMile
x_axis	$_	@X	@@N	String
thx1138	$plan9	@_	@@x_pos	MyClass
_26	$Global	@plan9	@@SINGLE	JazzSong

Sometimes creating arrays of words can be a pain, what with all the quotes and commas. Fortunately, Ruby has a shortcut: %w does just what we want:

Download **samples/intro_14.rb**

```
a = [ 'ant', 'bee', 'cat', 'dog', 'elk' ]
a[0]   # =>  "ant"
a[3]   # =>  "dog"
# this is the same:
a = %w{ ant bee cat dog elk }
a[0]   # =>  "ant"
a[3]   # =>  "dog"
```

Ruby hashes are similar to arrays. A hash literal uses braces rather than square brackets. The literal must supply two objects for every entry: one for the key, the other for the value. The key and value are normally separated by =>.

For example, you may want to map musical instruments to their orchestral sections. You could do this with a hash:

```
inst_section = {
  'cello'     => 'string',
  'clarinet'  => 'woodwind',
  'drum'      => 'percussion',
  'oboe'      => 'woodwind',
  'trumpet'   => 'brass',
  'violin'    => 'string'
}
```

The thing to the left of the => is the key, and the thing to the right is the corresponding value. Keys in a particular hash must be unique—you can't have two entries for "drum." The keys and values in a hash can be arbitrary objects—you can have hashes where the values are arrays, other hashes, and so on.

Hashes are indexed using the same square bracket notation as arrays. In this code, we'll use the p method to write the values to the console. This works like puts but displays values such as nil explicitly.

```
p inst_section['oboe']
p inst_section['cello']
p inst_section['bassoon']
```

produces:

```
"woodwind"
"string"
nil
```

As the previous example shows, a hash by default returns nil when indexed by a key it doesn't contain. Normally this is convenient, because nil means false when used in conditional expressions. Sometimes you'll want to change this default. For example, if you're using a hash to count the number of times each different word occurs in a file, it's convenient to have the default value be zero. Then you can use the word as the key and simply increment the corresponding hash value without worrying about whether you've seen that word before. This is easily done by specifying a default value when you create a new, empty hash. (The full source for the word frequency counter is on page 72.)

Download **samples/intro_17.rb**

```
histogram = Hash.new(0)    # The default value is zero
histogram['ruby']   # =>   0
histogram['ruby'] = histogram['ruby'] + 1
histogram['ruby']   # =>   1
```

Array and hash objects have lots of useful methods; see the discussion starting on page 67, and the reference sections starting on pages 447 and 533, for details.

Symbols

Often, when programming, you need to create a name for something significant. For example, you might want to refer to the compass points by name, so you'd write this:

```
NORTH = 1
EAST  = 2
SOUTH = 3
WEST  = 4
```

Then, in the rest of your code, you could use the constants instead of the numbers:

```
walk(NORTH)
look(EAST)
```

Most of the time, the actual numeric values of these constants are irrelevant (as long as they are unique). All you want to do is differentiate the four directions.

Ruby offers a cleaner alternative. *Symbols* are simply constant names that you don't have to predeclare and that are guaranteed to be unique. A symbol literal starts with a colon and is normally followed by some kind of name:

```
walk(:north)
look(:east)
```

There's no need to assign some kind of value to a symbol—Ruby takes care of that for you. Ruby also guarantees that no matter where it appears in your program, a particular symbol will have the same value. That is, you can write the following:

```
def walk(direction)
  if direction == :north
    # ...
  end
end
```

Symbols are frequently used as keys in hashes. We could write our previous example as this:

```
inst_section = {
  :cello     => 'string',
  :clarinet  => 'woodwind',
  :drum      => 'percussion',
  :oboe      => 'woodwind',
  :trumpet   => 'brass',
  :violin    => 'string'
}
inst_section[:oboe]     # =>   "woodwind"
inst_section[:cello]    # =>   "string"
# Note that strings aren't the same as symbols...
inst_section['cello']   # =>   nil
```

1.9

In fact, symbols are so frequently used as hash keys that Ruby 1.9 introduces a new syntax—you can use name: value pairs to create a hash if the keys are symbols:

```
inst_section = {
  cello:    'string',
  clarinet: 'woodwind',
  drum:     'percussion',
  oboe:     'woodwind',
  trumpet:  'brass',
  violin:   'string'
}
puts "An oboe is a #{inst_section[:oboe]}"
```

produces:

```
An oboe is a woodwind
```

Control Structures

Ruby has all the usual control structures, such as if statements and while loops. Java, C, and Perl programmers may well get caught by the lack of braces around the bodies of these statements. Instead, Ruby uses the keyword end to signify the end of a body:

```
if count > 10
  puts "Try again"
elsif tries == 3
  puts "You lose"
```

```
else
  puts "Enter a number"
end
```

Similarly, while statements are terminated with end:

```
while weight < 100 and num_pallets <= 30
  pallet = next_pallet()
  weight += pallet.weight
  num_pallets += 1
end
```

Most statements in Ruby return a value, which means you can use them as conditions. For example, the method gets returns the next line from the standard input stream or nil when end of file is reached. Because Ruby treats nil as a false value in conditions, you could write the following to process the lines in a file:

```
while line = gets
  puts line.downcase
end
```

Here, the assignment statement sets the variable line to either the next line of text or nil, and then the while statement tests the value of the assignment, terminating the loop when it is nil.

Ruby *statement modifiers* are a useful shortcut if the body of an if or while statement is just a single expression. Simply write the expression, followed by if or while and the condition. For example, here's a simple if statement:

```
if radiation > 3000
  puts "Danger, Will Robinson"
end
```

Here it is again, rewritten using a statement modifier:

```
puts "Danger, Will Robinson" if radiation > 3000
```

Similarly, a while loop such as this:

```
square = 2
while square < 1000
   square = square*square
end
```

becomes this more concise version:

```
square = 2
square = square*square  while square < 1000
```

These statement modifiers should seem familiar to Perl programmers.

Regular Expressions

Most of Ruby's built-in types will be familiar to all programmers. A majority of languages have strings, integers, floats, arrays, and so on. However, regular expression support is typically built into only scripting languages, such as Ruby, Perl, and awk. This is a shame, because regular expressions, although cryptic, are a powerful tool for working with text. And having them built in, rather than tacked on through a library interface, makes a big difference.

Entire books have been written about regular expressions (for example, *Mastering Regular Expressions* [Fri02]), so we won't try to cover everything in this short section. Instead, we'll look at just a few examples of regular expressions in action. You'll find full coverage of regular expressions starting on page 117.

A regular expression is simply a way of specifying a *pattern* of characters to be matched in a string. In Ruby, you typically create a regular expression by writing a pattern between slash characters (*/pattern/*). And, Ruby being Ruby, regular expressions are objects and can be manipulated as such.

For example, you could write a pattern that matches a string containing the text *Perl* or the text *Python* using the following regular expression:

```
/Perl|Python/
```

The forward slashes delimit the pattern, which consists of the two things we're matching, separated by a pipe character (|). This pipe character means "either the thing on the right or the thing on the left," in this case either *Perl* or *Python.* You can use parentheses within patterns, just as you can in arithmetic expressions, so you could also have written this pattern like this:

```
/P(erl|ython)/
```

You can also specify *repetition* within patterns. /ab+c/ matches a string containing an *a* followed by one or more *b*'s, followed by a *c*. Change the plus to an asterisk, and /ab*c/ creates a regular expression that matches one *a*, zero or more *b*'s, and one *c*.

You can also match one of a group of characters within a pattern. Some common examples are *character classes* such as \s, which matches a whitespace character (space, tab, newline, and so on); \d, which matches any digit; and \w, which matches any character that may appear in a typical word. A dot (.) matches (almost) any character. A table of these character classes appears on page 125.

We can put all this together to produce some useful regular expressions:

```
/\d\d:\d\d:\d\d/     # a time such as 12:34:56
/Perl.*Python/       # Perl, zero or more other chars, then Python
/Perl Python/        # Perl, a space, and Python
/Perl *Python/       # Perl, zero or more spaces, and Python
/Perl +Python/       # Perl, one or more spaces, and Python
/Perl\s+Python/      # Perl, whitespace characters, then Python
/Ruby (Perl|Python)/ # Ruby, a space, and either Perl or Python
```

Once you have created a pattern, it seems a shame not to use it. The match operator =~ can be used to match a string against a regular expression. If the pattern is found in the string, =~ returns its starting position; otherwise, it returns nil. This means you can use regular expressions as the condition in if and while statements. For example, the following code fragment writes a message if a string contains the text *Perl* or *Python*:

```
if line =~ /Perl|Python/
  puts "Scripting language mentioned: #{line}"
end
```

The part of a string matched by a regular expression can be replaced with different text using one of Ruby's substitution methods:

```
line.sub(/Perl/, 'Ruby')    # replace first 'Perl' with 'Ruby'
line.gsub(/Python/, 'Ruby') # replace every 'Python' with 'Ruby'
```

You can replace every occurrence of *Perl* and *Python* with *Ruby* using this:

```
line.gsub(/Perl|Python/, 'Ruby')
```

We'll have a lot more to say about regular expressions as we go through the book.

Blocks and Iterators

This section briefly describes one of Ruby's particular strengths. We're about to look at *code blocks*, which are chunks of code you can associate with method invocations, almost as if they were parameters. This is an incredibly powerful feature. One of our reviewers commented at this point: "This is pretty interesting and important, so if you weren't paying attention before, you should probably start now." We'd have to agree.

You can use code blocks to implement callbacks (but they're simpler than Java's anonymous inner classes), to pass around chunks of code (but they're more flexible than C's function pointers), and to implement iterators.

Code blocks are just chunks of code between braces or between do...end. This is a code block:

```
{ puts "Hello" }
```

So is this:

```
do
  club.enroll(person)
  person.socialize
end
```

Why are there two kinds of delimiter? It's partly because sometimes one feels more natural to write than another. It's partly too because they have different precedences: the braces bind more tightly than the do/end pairs. In this book, we try to follow what is becoming a Ruby standard and use braces for single-line blocks and do/end for multiline blocks.

All you can do with a block is associate it with a call to a method. You do this by putting the start of the block at the end of the source line containing the method call.

For example, in the following code, the block containing puts "Hi" is associated with the call to the method greet (which we don't show):

```
greet  { puts "Hi" }
```

If the method has parameters, they appear before the block:

```
verbose_greet("Dave", "loyal customer")  { puts "Hi" }
```

A method can then invoke an associated block one or more times using the Ruby yield statement. You can think of yield as being something like a method call that invokes the block associated with the call to the method containing the yield.

The following example shows this in action. We define a method that calls yield twice. We then call this method, putting a block on the same line, after the call (and after any arguments to the method).[3]

Download **samples/intro_41.rb**

```
def call_block
  puts "Start of method"
  yield
  yield
  puts "End of method"
end
call_block { puts "In the block" }
```

produces:

```
Start of method
In the block
In the block
End of method
```

The code in the block (puts "In the block") is executed twice, once for each call to yield.

You can provide arguments to the call to yield, and they will be passed to the block. Within the block, you list the names of the parameters to receive these arguments between vertical bars (| params... |). The following example shows a method calling its associated block twice, passing the block two arguments each time:

Download **samples/intro_42.rb**

```
def who_says_what
  yield("Dave", "hello")
  yield("Andy", "goodbye")
end
who_says_what {|person, phrase| puts "#{person} says #{phrase}"}
```

produces:

```
Dave says hello
Andy says goodbye
```

3. Some people like to think of the association of a block with a method as a kind of argument passing. This works on one level, but it isn't really the whole story. You may be better off thinking of the block and the method as coroutines, which transfer control back and forth between themselves.

Code blocks are used throughout the Ruby library to implement *iterators*, which are methods that return successive elements from some kind of collection, such as an array:

```
animals = %w( ant bee cat dog elk )    # create an array
animals.each {|animal| puts animal }   # iterate over the contents
```

produces:

```
ant
bee
cat
dog
elk
```

Many of the looping constructs that are built into languages such as C and Java are simply method calls in Ruby, with the methods invoking the associated block zero or more times:

Download **samples/intro_44.rb**

```
[ 'cat', 'dog', 'horse' ].each {|name| print name, " " }
5.times {  print "*" }
3.upto(6) {|i|  print i }
('a'..'e').each {|char| print char }
```

produces:

```
cat dog horse *****3456abcde
```

Here we ask an array to call the block once for each of its elements. Then, object 5 calls a block five times. Rather than use for loops, in Ruby we can ask the number 3 to call a block, passing in successive values until it reaches 6. Finally, the range of characters from *a* to *e* invokes a block using the method each.

Reading and 'Riting

Ruby comes with a comprehensive I/O library. However, in most of the examples in this book, we'll stick to a few simple methods. We've already come across two methods that do output: puts writes its arguments with a newline after each; print also writes its arguments but with no newline. Both can be used to write to any I/O object, but by default they write to standard output.

Another output method we use a lot is printf, which prints its arguments under the control of a format string (just like printf in C or Perl):

```
printf("Number: %5.2f,\nString: %s\n", 1.23, "hello")
```

produces:

```
Number:  1.23,
String: hello
```

In this example, the format string "Number: %5.2f,\nString: %s\n" tells printf to substitute in a floating-point number (with a minimum of five characters, two after the decimal point) and a string. Notice the newlines (\n) embedded in the string; each moves the output onto the next line.

You have many ways to read input into your program. Probably the most traditional is to use the routine gets, which returns the next line from your program's standard input stream:

```
line = gets
print line
```

Because gets returns nil when it reaches the end of input, you can use its return value in a loop condition. Notice that here the condition to the while is an assignment: we store whatever gets returns into the variable line and then test to see whether that returned value was nil or false before continuing:

```
while line = gets
  print line
end
```

Command-Line Arguments

When you run a Ruby program from the command line, you can pass in arguments. These are accessible in two different ways.

First, the array ARGV contains each of the arguments passed to the running program. Create a file called cmd_line.rb that contains the following:

```
puts "You gave #{ARGV.size} arguments"
p ARGV
```

When we run it with arguments, we can see that they get passed in:

```
$ ruby cmd_line.rb ant bee cat dog
```

produces:

```
You gave 4 arguments
["ant", "bee", "cat", "dog"]
```

Often, the arguments to a program are the names of files that you want to process. In this case, you can use a second technique: the variable ARGF is a special kind of I/O object that acts like all the contents of all the files whose names are passed on the command line (or standard input if you don't pass any filenames). We'll look at that some more on page 342.

Onward and Upward

That's it. We've finished our lightning-fast tour of some of the basic features of Ruby. We took a look at objects, methods, strings, containers, and regular expressions; saw some simple control structures; and looked at some rather nifty iterators. We hope this chapter has given you enough ammunition to be able to attack the rest of this book.

Time to move on and move up—up to a higher level. Next, we'll be looking at classes and objects, things that are at the same time both the highest-level constructs in Ruby and the essential underpinnings of the entire language.