



Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=417>



**MEAP Edition  
Manning Early Access Program**

Copyright 2008 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

## **Ruby Foundations Table of Contents**

### ***PART 1: RUBY FOUNDATIONS***

Chapter 1: Bootstrapping your Ruby literacy

Chapter 2: Objects, methods, and local variables

Chapter 3: Organizing objects with classes

Chapter 4: Modules and program organization

Chapter 5: The default object (self), scope, and visibility

Chapter 6: Controlling program flow

### ***PART 2: BUILT-IN CLASSES AND MODULES***

Chapter 7: Built-in essentials

Chapter 8: Strings, symbols, numbers, and other scalar objects

Chapter 9: Collections and containers

Chapter 10: Enumerables and enumerators

Chapter 11: Regular expressions and regex-based string operations

Chapter 12: File and IO operations

## ***PART 3: RUBY DYNAMICS***

Chapter 13: Objects and their individuality

Chapter 14: Callable and runnable objects

Chapter 15: Reflection and callbacks

## *preface*

In 2006, Manning published my book *Ruby for Rails: Ruby techniques for Rails developers*. My goal in writing *Ruby for Rails*--or, as it has come to be known, R4R--was to do what I could to ensure that Rails developers understood that as Rails developers they were, *ipso facto*, Ruby programmers, and that it was therefore worth learning Ruby well. Ruby topics were chosen for inclusion in, or exclusion from, *Ruby for Rails* based on my judgment as to their likely importance to Rails developers. The fact that it was *my* judgment meant that they weren't always the topics that every Rails developer had already heard of; and that was, to a large extent, the point.

I'm happy to say that critical response to R4R was good. Complaints came mostly from people who didn't get it (or, as some well-wishes have quipped, people who didn't even bother reading the title), and criticized the book for doing too precise a job of delivering what it said it would deliver: Ruby, optimized for consumption by Rails developers but still Ruby. People who understood the book's agenda (and I'm happy to say that that appears to include most readers) gave the book a warm reception.

One consistent refrain of praise and encouragement took the form of describing *Ruby for Rails* as not only a good Ruby treatment for Rails developers, but a good Ruby book for anyone interested in Ruby. *Why*, I was asked on a number of occasions, *don't you write a just plain Ruby book, not just for Rails people but for anyone interested in Ruby?*

And so, to make a long story short, I have. *The Well-Grounded Rubyist* is a "just plain Ruby" book, for anyone interested in Ruby. It's a revision of *Ruby for Rails* or, one might say, a repurposing. I don't mean for Rails developers *not* to read it; in fact, I'm cautiously optimistic that in the couple of years since R4R was published, the idea that Rails developers should learn Ruby has become a commonplace, and many people who first got into Ruby through Rails have gotten interested in Ruby in its own right.

There's a lot of overlap with *Ruby for Rails*, but there's also a lot of new material, and everything has been oiled and polished and spiffed up to work with Ruby 1.9. In honor of the fact that 1.9.0 had already been released when I started working on the revisions in earnest, I don't even maintain a running commentary about what's new (though I do point out some things that I think people might otherwise stumble on). This is a Ruby 1.9 book.

Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

And I hope you enjoy it.

Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

## about this book

### Welcome...

...to *The Well-Grounded Rubyist*. This book is a reworking of my book *Ruby for Rails* (Manning, 2006), for the purpose of leveraging and building upon the Ruby language explication present in that book so that a broader audience of Ruby learners can make use of it and learn from it.

### How this book is organized

*The Well-Grounded Rubyist* consists of XX chapters and is divided into three parts. [MORE ON THIS.]

Ruby is a system; its component parts are not separate, and it's always a challenge to present a non-linear system in a linear format. Nor is there one and only one way to do it; different authors writing about the same language present it very differently, and that's as it should be. Each of us has instincts and experiences that tell us how we can most effectively put the topic across.

My instinct and experience, as well as my taste as a reader, tells me that one good barometer of how well I've ordered the march of topics is the number of forward references I have to make--with fewer being better. You can't eliminate them entirely, because there are circular topic dependencies in Ruby. So I accept that there will be some, but I try to keep them to a minimum. I really don't like to see (or write) more than I have to along the lines of, "As you'll see in Chapter <3 chapters from now>...."

One way to keep forward references to a reasonable minimum is to make a virtue of necessity: Take a bunch of forward references, and turn them into a chapter or section. That's what I do in Chapter 1; I give you a Ruby literacy guide, which consists entirely of Ruby techniques that you'll get to know in much greater depth later, but that will bootstrap you into understanding the treatment of the topics as they come along. Something similar happens in Chapter 7, where you get an infusion of information about how Ruby's built-in classes behave, and what they have in common, so that when we start exploring those classes in more depth in the ensuing chapters, you won't have to learn all the little things at the same time.

Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

A book is, inevitably, linear; but I think of the learning process as describing a spiral. You make a pass through a topic, or a landscape of topics; then you come back, perhaps at closer range, or at a more deliberate pace, and fill things in. That pass, in turn, serves as a first pass through yet more topics and points; and then you come back to those, and so forth. *The Well-Grounded Rubyist* is organized with that principle in mind.

### **Who should read this book**

*The Well-Grounded Rubyist* is optimized for a reader who's done some programming, and perhaps even some Ruby, and wants to learn more about the Ruby language--not only the specific techniques (though plenty of those), but also the design principles that make Ruby what it is. I'm a great believer in knowing what you're doing. Knowing what you're doing doesn't mean you have to compose a treatise in your head every time you write a line of code. It means that you know how to make the most out of the language, and you understand how to analyze problems when they arise.

I've hedged my bets a little, in terms of targeted readership, in that I've included some introductory remarks about a number of topics and techniques that are quite possibly familiar to experienced programmers. I ask the indulgence of those readers. The remarks in question go by pretty quickly, and I believe that even a few words of explanation of terms here and there can make a surprisingly big difference in how many people feel at home in, and welcomed by, the book. So if you see a passage here or there where I seem to be spoon-feeding a bit, please bear with it. It's in a good cause.

### **What this book doesn't include**

*The Well-Grounded Rubyist* is a serious, extensive look at the Ruby language. But it is not a reference work. There are core classes that I say little or nothing about, and I only discuss a small number of standard library extensions. That's by design. You really don't need me to spell out for you how to use every standard-library API, and I don't. What you do need, in all likelihood, is someone to explain to you what `class << self` really means, or why two instance variables two lines away from each other aren't the same variable, or what an Enumerator is and how it differs from an Iterator. I'm not opposed to learning how to program the standard library--far from it; you should and must--but that's not on this particular book's agenda.

### **A word on Ruby versions**

The current state of Ruby versions can be a bit perplexing. (And I'm not even talking about implementations; I'm talking strictly about versions of what has come to be known as *MRI*, the Matz Ruby interpreter, which is what is generally meant by "Ruby.")

There's Ruby 1.8.6, which I think of as the last full-blooded version of Ruby 1.8. Big changes came along in Ruby 1.9--but also a lot of issues about stability and readiness for production use. I don't think I know anyone who switched over completely to Ruby 1.9.0 when it came out. Ruby 1.9.1 is a more mature, more stable version of Ruby 1.9.

Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

Then there's version 1.8.7, which is technically 1.8 (it doesn't run on the 1.9 VM; it uses the "old" version of the interpreter) but which contains a huge number of backports from 1.9. It has a very 1.9-ish feel to it, and it exists, as I understand it, principally to facilitate migration to 1.9. I'm not convinced that it serves this purpose. I don't use it, because I like 1.8.6 and when I migrate to 1.9 completely I expect I'll just do it all in one step.

The impact of all of this on this book is as follows. The center of gravity is firmly on Ruby 1.9.1. When I refer back to 1.8, I always refer to 1.8.6, because that's where the biggest differences with 1.9 are to be found. I do not discuss 1.8.7, because it's a hybrid version and documenting everything it does and does not do would add disproportionately to the text. The best advice I can give with regard to 1.8.7 is to try things out. Anything I mention as being in 1.8.6 *or* 1.9 may or may not work in 1.8.7.

## **Code conventions, examples, and downloads**

In the text, names of Ruby variables and constants are in monospace. Names of classes and modules are in monospace where they represent direct references to existing class or module objects; for example, "Next, we'll reopen the class definition block for Composer." Where the name of a class or module is used in a more high-level narrative sense, the name appears in regular type; for example, "The domain will include a Composer class." In all cases, you'll be able to tell from the context that a class, module, or other Ruby entity is under discussion.

Names of directories and files are in monospace. Names of programs, such as ruby and rails, are in monospace where reference is made directly to the program executable or to command-line usage; otherwise, they appear in regular type.

Names of relational database tables and fields appear in *italics*.

Technical terms, on first mention, appear in *italics*. Italics are used for wildcard expressions, such as entity\_controller.rb, which indicates a file name with an "entity" component plus an underscore and the remaining text. A matching filename would be, for example, composer\_controller.rb.

The standalone code samples in the book can be run either by placing them in a text file and running the ruby command on them, or by typing them into the interactive Ruby interpreter irb. (Both of these techniques are explained in chapter 1.) Toward the beginning of the book, you'll be walked through the process of creating and naming program files and saving code samples in them. As the book progresses, it will assume that you can do this on your own. Only if it really matters—including, of course, in connection with the actual Rails applications you'll develop—will specific filenames for examples be suggested after the first few.

A considerable number of examples in the book, particularly in part 3 (Ruby built-ins), are presented in the form of irb (Interactive Ruby) sessions. What you'll see on the page are cut-and-pasted lines from a live interactive session, where the code was entered into irb and irb responded by running the code. You'll be alerted the first few times this format is used and when it reappears after a hiatus. You'll also come to recognize it easily (especially if you

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

start using irb). This mode of presentation is particularly suitable for short code snippets and expressions; and because irb always prints out the results of executing whatever you type in (rather like a calculator), it lets you see results while economizing on explicit print commands.

In other cases, the output from code samples is either printed separately after the samples, printed alongside the code (and clearly labeled as "output"), or embedded in the discussion following the appearance of the code.

Some examples are accompanied by numbered cueballs that appear to the side of the code. These cueballs are linked to specific points in the ensuing discussion and give you a way to refer back quickly to the line to which the discussion refers.

Command-line program invocations are shown with a dollar-sign (\$) prompt, in the general style of shell prompts in UNIX-like environments. The commands will work on Windows, even though the prompt may be different. (In all environments, the availability of the commands depends, as always, on the setting of the relevant path environment variable.)

The use of "web" rather than "Web" to designate the World Wide Web is a Manning in-house style convention which I have followed here, though in other contexts I follow the W3C's guideline, which is to use "Web."

# Part 1: Ruby Foundations

The goal of this part of the book is to give you a broad but practical foundation layer on which to build, and to which to anchor, the further explorations of Ruby that follow in parts 2 and 3. We'll start with a chapter on bootstrapping your Ruby literacy; after working through that first chapter, you'll be able to run Ruby programs comfortably and have a good sense of the layout of a typical Ruby installation. Starting with chapter 2 we'll get into the details of the Ruby language. Ruby is an object-oriented language, and the sooner you dive into how Ruby handles objects, the better. Accordingly, objects will serve both as a way to bootstrap the discussion of the language (and your knowledge of it), and as a golden thread leading us to further topics and techniques.

Objects are created by classes, and in chapter 3 you'll learn how classes work. The discussion of classes is followed by a look at modules in chapter 4. Modules allow you to fine-tune classes and objects by splitting out some of the object design into separate, reusable units of code. In order to understand Ruby programs--both your own and others'--you need to know about Ruby's notion of a "default object," known by the keyword *self*; and chapter 5 will take you deep into the concept of self, along with a treatment of Ruby's handling of variable visibility and scope.

In chapter 6, the last in this part of the book, you'll learn about control flow in Ruby programs--that is, how to steer the Ruby interpreter through conditional (*if*) logic, how to loop repeatedly through code, and even how to break away from normal program execution when an error occurs. By the end of chapter 6, you'll be thinking along with Ruby as you write and develop your code.

The title of this part is "Ruby Foundations," which obviously suggests that what's here is to be built upon later. And that's true. But it doesn't mean that the material in part 1 isn't important in itself. As you'll see once you start them, these six chapters present you with real Ruby techniques, real code, and information you'll use every time you write or execute a Ruby program. It's the "foundations," not because you'll learn it once and then ignore it, but because there's so much *more* about Ruby yet to follow!

# 1

## *Bootstrapping your Ruby literacy*

In this chapter

- A Ruby syntax survival kit
- Tour of the Ruby installation
- Walk-throughs of sample Ruby programs
- The mechanics of Ruby extensions
- Ruby's out-of-the-box command-line tools

This book will give you a foundation in Ruby, and this chapter will give your foundation a foundation. The goal of the chapter is to bootstrap you into the study of Ruby with enough knowledge and skill to proceed comfortably into what lies beyond.

We're going to look at basic Ruby syntax and techniques, and at how Ruby works: what you do when you write a program, how you get Ruby to run your program, and how you split a program into more than one file. You'll learn several variations on the process of running the Ruby interpreter (the program with the actual name `ruby`, to which you feed your program files for execution) as well how to use some important auxiliary tools designed to make your life as a Rubyist easier and more productive.

The chapter is based on a view of the whole Ruby landscape as being divided into three fundamental levels:

- The core language: design principles, syntax, semantics
- The extensions and libraries that ship with Ruby, and the facilities for adding extensions of your own
- The command-line tools that come with Ruby, with which you run the interpreter as well as some other important utilities.

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

It's not always possible to move in a strictly linear way through these three levels or topic area; after all, they're interlocking parts of a single system. But we'll get as close to linear in this chapter as is reasonably possible; and you can, in any case, use the three level descriptions as pegs to hang subtopics on, wherever they're introduced.

Nor does this first chapter exist solely in the service of later chapters. It has content in its own right: you'll learn real Ruby techniques and important points about the design of the language. The goal is to bootstrap or jump-start you, but even that process will involve close examination of some key aspects of the Ruby language.

### **Ruby, ruby, and ... RUBY?!**

Ruby is a programming language. We talk about things like “learning Ruby,” and we ask questions like, “Do you know Ruby?” The lowercase version, `ruby`, is a computer program; specifically, it's the Ruby interpreter, the program that reads your programs and runs them. You'll see this name used in sentences like, “I ran `ruby` on my file, but nothing happened,” or “What's the full path to your `ruby` executable?” Finally, there's RUBY—or, more precisely, there isn't. Ruby isn't an acronym, and it's never correct to spell it in all capital letters. People do this, as they do (also wrongly) with Perl, perhaps because they're used to seeing language names like BASIC and FORTRAN. Ruby isn't such a language. It's Ruby for the language, `ruby` for the interpreter.

## **1.1 Basic Ruby language literacy**

The goal of this section is to get you going with Ruby. It takes a breadth-first approach: we'll walk through the whole cycle of learning some syntax, writing some code, and running some programs.

At this point, you need to have Ruby installed on your computer. You also need a text editor (you can use any editor you like, as long as it's a plain-text editor and not a word processor) and a directory (a.k.a. folder) in which to store your Ruby program files. You might name that directory `rubycode` or `rubysamples`—any name is fine, as long as it's separate from other work areas so that you can keep track of your practice program files.

Before we get to program files, though, we'll take a first (but not last!) look at the interactive Ruby console program `irb`.

### **1.1.1 Meet Interactive Ruby (`irb`), your new best friend**

The `irb` utility ships with Ruby and is the most widely used Ruby command-line tool other than the interpreter itself. After starting `irb`, you type Ruby code into it, and it executes the code and prints out the resulting value.

Because `irb` is one of the command-line tools that ship with Ruby, it's not discussed in detail until section 1.4.2. Feel free to jump to that section and have a look; it's pretty straightforward. Or, type `irb` at the command line, and enter sample code as you encounter it in the text. Having an open `irb` session means you can test Ruby snippets any time and in

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

any quantity. Most Ruby developers find `irb` indispensable, and you'll see a few examples of its use as we proceed through this chapter.

The `irb` examples you'll see in this book will use a command-line option that makes `irb` output a little easier to read:

```
irb --simple-prompt
```

If you want to see the effect of the `--simple-prompt` option, try starting `irb` with it and without it. As you'll see, the simple prompt keeps your screen a lot clearer. The default (non-simple) prompt displays more information, such as a line-number count for your interactive session; but for the examples we'll be looking at, the simple prompt is sufficient.

Now let's continue to bootstrap your Ruby literacy so we have a shared ground on which to continue building and exploring. One thing you'll need is enough exposure to basic Ruby syntax to get you started.

### 1.1.2 A Ruby syntax survival kit

Table 1.1 summarizes some Ruby techniques that you'll find useful in understanding the examples in this chapter, and in starting to experiment with Ruby yourself. These techniques are summarized in the table for you to peruse in advance and easily refer back to later.

Table 1.1 Synopsis of key elements of Ruby syntax for Ruby literacy bootstrapping purposes

Operation	Example(s)	Comments
Arithmetic	<code>2 + 3</code> (addition)	All these operations work on integers or floating-point numbers ("floats"). Mixing integers and floats together, as some of the examples do, produces a floating-point result.  Note that you need to write <code>0.23</code> , rather than <code>.23</code> .
	<code>2 - 3</code> (subtraction)	
	<code>2 * 3</code> (multiplication)	
	<code>2 / 3</code> (division)	
	<code>10.3 + 20.25</code>	
	<code>103 - 202.5</code>	
	<code>32.9 * 10</code>	
	<code>100.0 / 0.23</code>	
Assignment	<code>x = 1</code>	Binds a local variable (on the left) to a value (on the right).
	<code>string = "Hello"</code>	

Print something to the screen	<pre>print "Hello" puts "Hello"</pre>	<p><code>puts</code> adds a newline to the string it outputs, if there isn't one at the end already. <code>print</code> doesn't.</p>
	<pre>x = "Hello" puts x</pre>	
	<pre>x = "Hello" print x</pre>	<p><code>print</code> prints exactly what it's told to and leaves the cursor at the end. (Note: on some platforms, an extra newline is automatically output at the end of a program.)</p>
	<pre>x = "Hello" p x</pre>	<p><code>p</code> outputs an inspect string, which may contain extra information about what it's printing.</p>
Get a line of keyboard input	<pre>gets string = gets</pre>	<p>You can assign the input line directly to a variable (the variable <code>string</code> in the second example).</p>
Convert a numeric string to a number	<pre>x = "100".to_i s = "100" x = s.to_i</pre>	<p>To perform arithmetic, you have to make sure you have numbers rather than strings of characters. <code>to_i</code> performs string-to-integer conversion.</p>
Compare two values	<pre>x == y</pre>	<p>Note the two equal signs (not just one, as in assignment).</p>
Conditional execution	<pre>if x == y   puts "Yes!" else   puts "No!" end</pre>	<p>Conditional statements always end with the word <code>end</code>.</p>

Special value objects	<code>true</code> <code>false</code> <code>nil</code>	The objects <code>true</code> and <code>false</code> often serve as return values for conditional expressions. The object <code>nil</code> is a kind of “non-object”, indicating absence of a value or result. <code>false</code> and <code>nil</code> will cause a conditional expression to fail; all other objects (including <code>true</code> , of course, but also including <code>0</code> and empty strings) will cause it to succeed.
The default object	<code>self</code>	The keyword <code>self</code> refers to the default object. <code>self</code> is a role that different objects play, depending on the execution context. Method calls that do not specify an object are called on <code>self</code> .
Put comments in code files	<code># This is a comment line.</code> <code>X = 1 # Comment after code</code>	Comments are ignored by the interpreter.

A few fundamental aspects of Ruby and Ruby syntax are too involved for summary in a table. You need to be able to recognize a small handful of different Ruby identifiers and, above all, you need a sense of what an object is in Ruby and what a method call looks like. We'll take a first look at both of those aspects of the language next.

### **1.1.3 The variety of Ruby identifiers**

Ruby has a small number of identifier types that you'll want to be able to spot and differentiate from each other at a glance. The identifier family tree looks like this:

- Variables
  - local
  - instance
  - class
  - global
- Constants
- Keywords
- Method names

Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

It's a small family, and easily learned. We'll survey them here. Keep in mind that this section's purpose is to teach you to recognize the various identifiers. You'll also be learning a lot more, throughout the book, about when and how to use them. This is just the first lesson in identifier literacy.

### VARIABLES

**Local variables** start with a lowercase letter or an underscore, and consist of lowercase letters, underscores, and/or digits. `x`, `string`, `__abc__`, `person2` are all valid local variable names. It's customary to use underscores, rather than camelcase, when composing local variable names from multiple words: `first_name`, rather than `firstName`.

**Instance variables**, which serve the purpose of storing information for individual objects, always start with a single at-sign (`@`), and consist thereafter of the same character set as local variables. While a local variable cannot start with an uppercase letter, an instance variable can have one in the first position after the at-sign (though it may not have a digit in this position).

**Class variables**, which store information per class hierarchy (again, don't worry about the semantics at this stage), follow the same rules as instance variables, except that they start with *two* at-signs: `@@example`.

**Global variables** are recognizable by their leading dollar sign: `$example`. The segment after the dollar sign does not follow local-variable naming conventions; there are global variables called `$:`, `$1`, and `$/`, as well as `$stdin` and `$LOAD_PATH`. As long as it begins with a dollar sign, it's a global variable. As for the non-alphanumeric ones, the only such identifiers you're like to see are predefined ones, so you don't need to worry about which punctuation marks are legal and which aren't.

### CONSTANTS

Constants begin with an uppercase letter. `A`, `String`, `FirstName`, and `STDIN` are all valid constant names. It's customary to use either camelcase (`FirstName`) or underscore-separate all-uppercase words (`FIRST_NAME`) in composing constant names from multiple words.

### KEYWORDS

Ruby has numerous keywords: pre-defined, reserved terms associated with specific programming tasks and contexts. Keywords include `def` (for method definitions), `class` (for class definitions), `if` (conditional execution), and `__FILE__` (the name of the file currently being executed). There are about 40 of them, and they're generally short, single-word (as opposed to composite) identifiers.

### METHOD NAMES

Names of methods in Ruby follow the same rules, and the same conventions, as local variables. This is by design: methods do not call attention to themselves as methods, but rather blend into the texture of a program as, simply, expressions that provide a value. There are contexts in which you cannot tell, just by looking at an expression, whether you're seeing a local variable or a method name—and that's intentional.

Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

Speaking of methods: now that you've got a roadmap to Ruby identifiers, let's get back to some language semantics, in particular the all-important role of the object.

### 1.1.4 Method calls, messages, and Ruby objects

Ruby sees all data structures and values, including scalar (atomic) values like integers and strings, but also including complex data structures like arrays, as *objects*. Every object is capable of understanding a certain set of *messages*. Each message that an object understands corresponds directly to a *method*: a named, executable routine whose execution the object has the ability to trigger.

Objects are represented either by literal constructors—like quotation marks for strings— or by variables to which they have been bound. Message-sending is achieved via the special dot operator: the message to the right of the dot is sent to the object on the left of the dot. (There are other, more specialized ways to send messages to objects, but the dot is the most common, most fundamental way.) In this example from table 1.1

```
x = "100".to_i
```

the dot means that the message "to\_i" is being sent to the string "100". The string "100" is called the *receiver* of the message. We can also say that the method `to_i` is being *called on* the string "100". The example also illustrates the binding of the result—the number 100—to the variable `x`.

#### Why the double terminology?

Why bother saying both "sending the message 'to\_i'" and "calling the method `to_i`"? Why have two ways of describing the same operation? Because they aren't quite the same. Most of the time, you send a message to a receiving object, and the object executes the corresponding method. But sometimes, there is no corresponding method. You can put anything to the right of the dot, and there's no guarantee that the receiver will have a method that matches the message you send.

If that sounds like chaos, it isn't, because objects can intercept unknown messages and try to make sense of them. The Ruby on Rails Web development framework, for example, makes heavy use of the technique of sending unknown messages to objects, intercepting those messages, and making sense of them on the fly based on dynamic conditions like the names of the columns in the tables of the current database.

Methods can take *arguments*, which are also objects. (Almost everything in Ruby is an object, although there are some syntactic structures that help you create and manipulate objects but aren't, themselves, objects.) Here's a method call with an argument:

```
x = "100".to_i(9)
```

Calling `to_i` on 100 with an argument of 9 generates a decimal integer equivalent to the base-nine number 100: `x` is now equal to 81 decimal.

This example also shows the use of parentheses around method arguments. These parentheses are usually optional, but in more complex cases they may be required to clear

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

up what might otherwise be ambiguities in the syntax. Many programmers use parentheses in most or all method calls, just to be safe (and for visual clarity).

The whole universe of a Ruby program consists of objects and the messages that are sent to them. As a Ruby programmer, you spend most of your time either specifying the things you want objects to be able to do (by defining methods) or asking the objects to do those things (by sending them messages).

We'll explore all of this in much greater depth later in the book. Again, this brief sketch is just for Ruby literacy bootstrapping purposes. When you see a dot in what would otherwise be an inexplicable position, you should interpret it as a message (on the right) being sent to an object (on the left). Keep in mind, too, that some method calls take the form of *bareword*-style invocations, like the call to `puts` in this example:

```
puts "Hello."
```

Here, in spite of the lack of a message-sending dot and an explicit receiver for the message, we're sending the message "puts" with the argument "Hello." to an object: the "default object" `self`. There's always a self defined when your program is running, though *which* object is `self` changes, according to very specific rules. You'll learn much more about `self` in chapter 5. For now, take note of the fact that a bareword like "puts" can be a method call.

The most important concept in Ruby is the concept of the object. Closely related, and playing an important supporting role, is the concept of the *class*.

#### **THE ORIGIN OF OBJECTS IN CLASSES**

Classes define clusters of behavior or functionality, and every object is an instance of exactly one class. Ruby provides a large number of built-in classes, representing important foundational data types (classes like `String`, `Array`, `Fixnum`). Every time you create a string object, you've created an instance of the class `String`.

You can also write your own classes. You can even modify existing Ruby classes; if you don't like the way strings or arrays behave, you can change it. It's almost always a bad idea to do so, but Ruby does allow it. (We'll look at the pros and cons of making changes to built-in classes in chapter 13.)

Although every Ruby object is an instance of a class, the concept of class is less important than the concept of object. That's because objects can change, acquiring methods and behaviors that were not defined in their class. The class is responsible for "launching" the object, but the object, thereafter, has a life of its own.

The ability of objects to adopt behaviors that their class didn't give them is one of the most central defining principles of the design of Ruby as a language. As you can surmise, we'll be coming back to it frequently, in a variety of contexts. For now, just be aware that while every object has a class, the class of an object is not the sole determinant of what the object can do.

Armed with some Ruby literacy (and a summary to refer back to when in doubt), let's walk through the steps involved in running a program.

### 1.1.5 Writing and saving a sample program

At this point, you can start creating program files in the Ruby sample code directory you created a little while back. Our first program will be a Celsius-to-Fahrenheit temperature converter. We'll walk this example through several stages, adding to it and modifying it as we go. The first version is simple; the focus is on the file-creation and program-running processes, rather than any elaborate program logic.

#### NOTE: INTEGER TEMPERATURES

In the examples that follow, all the arithmetic involves integers, rather than floating-point numbers. Ruby handles both, of course; and if you were writing a “real-world” temperature converter you'd want to use floating-point numbers. We're sticking to integers here, in the input and the output, for the sake of simplicity and so as to keep the focus on matters of program structure and execution.

#### CREATING A FIRST PROGRAM FILE

Using a plain-text editor, type the code from listing 1.1 into a text file, and save it under the filename `c2f.rb` in your sample code directory.

#### Listing 1.1 Simple, limited-purpose Celsius-to-Fahrenheit converter (`c2f.rb`)

```
celsius = 100
fahrenheit = (celsius * 9 / 5) + 32
puts "The result is: "
puts fahrenheit
puts "."
```

You now have a complete (albeit tiny) Ruby program on your disk, and you can run it.

#### NOTE

Depending on your operating system, you may be able to run Ruby program files standalone—that is, with just the filename, or with a short name (like `c2f`) and no file extension. Keep in mind, though, that the `.rb` filename extension is mandatory in some cases, mainly involving programs that occupy more than one file (which you'll learn about in detail later) and that need a mechanism for the files to find each other. In this book, all Ruby program filenames end in `.rb` to ensure that the examples work on as many platforms, and with as few administrative digressions, as possible.

### 1.1.6 Feeding the program to Ruby

Running a Ruby program involves passing the program's source file (or files) to the Ruby interpreter, which is called `ruby`. We'll do that now ... sort of. We'll feed the program to `ruby`; but instead of asking Ruby to run the program, we'll just ask it to check the program code for syntax errors.

Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

**CHECKING FOR SYNTAX ERRORS**

If you add 31 instead of 32 in your conversion formula, that's a programming error. Ruby will still happily run your program and give you the flawed result. But if you accidentally leave out the closing parenthesis in the second line of the program, that's a syntax error, and Ruby won't run the program:

```
$ ruby broken_c2f.rb
broken_c2f.rb:5: syntax error, unexpected $end, expecting ')'
```

(The error is reported on line 5—the last line of the program—because Ruby waits patiently to see whether you're ever going to close the parenthesis before concluding that you're not.)

Conveniently, the Ruby interpreter can check programs for syntax errors without running the programs. It reads through the file and tells you whether the syntax is OK. To run a syntax check on your file, do this:

```
$ ruby -cw c2f.rb
```

The `-cw` command-line flag is shorthand for two flags: `-c` and `-w`. The `-c` flag means *check for syntax errors*. The `-w` flag activates a higher level of *warning*; Ruby will fuss at you if you've done things that are legal Ruby but are questionable on grounds other than syntax.

```
Assuming you've typed the file correctly, you should see the message
Syntax OK
```

printed on your screen.

**RUNNING THE PROGRAM**

To run the program, you pass the file once more to the interpreter, but this time without the combined `-c` and `-w` flags:

```
$ ruby c2f.rb
```

If all goes well, you'll see the output of the calculation:

```
The result is
212
.
```

The result of the calculation is correct, but the output, spread as it is over three lines, looks bad. We want it all on one line.

**A SECOND CONVERTER ITERATION**

The problem can be traced to the difference between the `puts` command and the `print` command. `puts` adds a newline to the end of the string it prints out, if the string doesn't end with one already. `print`, on the other hand, prints out the string you ask it to and then stops; it doesn't automatically jump to the next line.

```
To fix the problem, you can change the first two puts commands to print:
print "The result is "
print fahrenheit
puts "."
```

(Note the blank space after `is`, which ensures that there will be a space between `is` and the number.) Now the output is as follows:

```
The result is 212.
```

`puts` is short for *put* (that is, print) *string*. Although *put* may not intuitively invoke the notion of skipping down to the next line, that's what `puts` does: like `print`, it prints what

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

you tell it to, but then it also automatically goes to the next line. If you ask `puts` to print a line that already ends with a newline, it doesn't bother adding one.

If you're used to print facilities in languages that don't automatically add a newline (such as Perl's `print` function), you may find yourself writing code like this in Ruby when you want to print a value followed by a newline:

```
print fahrenheit, "\n"
```

You almost never have to do this, though, because `puts` adds a newline for you. You'll pick up the `puts` habit, along with other Ruby idioms and conventions, as you go along.

### WARNING

On some platforms (Windows, in particular), an extra newline character is printed out at the end of the run of a program. This means a `print` that should really be a `puts` will be hard to detect, because it will act like a `puts`. Being aware of the difference between the two, and using the one you want based on the usual behavior, should be sufficient to ensure you get the desired results.

Having looked a little at screen output, let's widen the I/O field a bit to include keyboard input and file operations.

### 1.1.7 Keyboard and file I/O

Ruby offers lots of techniques for reading data during the course of program execution, both from the keyboard and from disk files. You'll find uses for them—if not in the course of writing every application, then almost certainly while writing Ruby code to maintain, convert, housekeep, or otherwise manipulate the environment in which you work. We'll look at some of these input techniques here; an expanded look at I/O operations can be found in Chapter 12.

#### KEYBOARD INPUT

A program that tells you over and over again that 100° Celsius is 212° Fahrenheit has limited value. A more valuable program lets you specify a Celsius temperature and tells you the Fahrenheit equivalent.

Modifying the program to allow for this functionality involves adding a couple of steps and using two methods from table 1.1: `gets` (get a line of keyboard input) and `to_i` (convert to an integer). (one of which you're familiar with already). Because this is a new program, not just a correction, put the version from listing 1.2 in a new file (`c2fi.rb`; *i* stands for interactive):

#### Listing 1.2 Interactive temperature converter (c2fi.rb)

```
print "Hello. Please enter a Celsius value: "
celsius = gets
fahrenheit = (celsius.to_i * 9 / 5) + 32
print "The Fahrenheit equivalent is "
print fahrenheit
puts "."
```

Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

A couple of sample runs demonstrate the new program in action:

```
$ ruby c2fi.rb
Hello. Please enter a Celsius value: 100
The Fahrenheit equivalent is 212.
$ ruby c2fi.rb
Hello. Please enter a Celsius value: 23
The Fahrenheit equivalent is 73.
```

### Shortening the code

You can shorten the program considerably by consolidating the operations of input, calculation, and output. A compressed rewrite looks like this:

## Following code lines are part of the sidebar

```
print "Hello. Please enter a Celsius value: "

print "The Fahrenheit equivalent is ", gets.to_i * 9 / 5 + 32,
      ".\n"
```

This version economizes on variables—there aren't any—but requires anyone reading it to follow a somewhat denser (but shorter!) set of expressions. Any given program usually has several or many spots where you have to decide between longer (but maybe clearer?) and shorter (but perhaps a bit cryptic...). And sometimes, shorter can be clearer. It's all part of developing a Ruby coding style.

### READING FROM A FILE

Reading a file from a Ruby program isn't much more difficult, at least in many cases, than reading a line of keyboard input. Here, we'll read one number from a file and convert it from Celsius to Fahrenheit.

First, create a new file called `temp.dat` (temperature data), containing just one line with one number on it:

```
100
```

Now, create a third program file, called `c2fin.rb` (*in* for [file] input), as shown in listing 1.3.

### Listing 1.3 Temperature converter using file input (c2fin.rb)

```
puts "Reading Celsius temperature value from data file..."
num = File.read("temp.dat")
celsius = num.to_i
fahrenheit = (celsius * 9 / 5) + 32
puts "The number is " + num
print "Result: "
puts fahrenheit
```

This time, the sample run and its output look like this:

```
$ ruby c2fin.rb
Reading Celsius temperature value from data file...
The number is 100
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

Result: 212

Naturally, if you change the number in the file, the result will be different.

What about writing the result of the calculation *to* a file?

### WRITING TO A FILE

The simplest file-writing operation is just a little more elaborate than the simplest file-reading operation. As you can see from listing 1.4, the main extra step when you write to a file is the specification of a file *mode*—in this case, *w* (for *write*).

Save the version of the program from listing 1.4 to `c2fout.rb`, and run it.

#### Listing 1.4 Temperature converter with file output (`c2fout.rb`)

```
print "Hello. Please enter a Celsius value: "
celsius = gets.to_i
fahrenheit = (celsius * 9 / 5) + 32
puts "Saving result to output file 'temp.out'"
fh = File.new("temp.out", "w")
fh.puts fahrenheit
fh.close
```

The method-call `fh.puts fahrenheit` has the effect of printing the value of `fahrenheit` to the file for which `fh` is an output handle. If you inspect the file `temp.out`, you should see that it contains the Fahrenheit equivalent of whatever number you typed in.

As an exercise, you might try to combine the previous examples into a Ruby program that reads a number from a file and writes the Fahrenheit conversion to a different file. Meanwhile, with some basic Ruby syntax in place, our next stop will be an examination of the Ruby installation. This, in turn, will equip us for a look at how Ruby manages extensions and libraries.

## 1.2 Anatomy of the Ruby installation

Installing Ruby on your system means installing numerous components of the language, possibly including the source code for the language, and definitely including a number of disk directories' worth of Ruby-language libraries and support files. You won't necessarily use everything mentioned in this section every time you write something in Ruby, but it's good to know what's there. Also, quite a few of the programming libraries that come bundled with Ruby are written in Ruby—so knowing your way around the Ruby installation will enable you to look at some well-written Ruby code and (we hope) absorb some good habits.

We'll look at several of the subdirectories of the main Ruby installation to give you a general sense of what's in them. This is just an overview. The best way—really, the only way—to get to know the Ruby installation layout and become comfortable with it is to navigate around it and see what's there.

### TIP

You may already have the Ruby source code tree on your machine, and if not, you can download it from the Ruby homepage. The source code tree contains a lot of Ruby files

Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

that end up in the eventual installation, and it also contains a lot of C files that get compiled into object files that then get installed. In addition, the source tree contains informational files like the ChangeLog and software licenses.

Before you can either navigate generally or pinpoint files specifically, you need to know where Ruby is installed on your system. The best way to find out is to ask Ruby, via `irb`. You'll want to preload a Ruby library package called `rbconfig` into your `irb` session. `rbconfig` is an interface to a lot of compiled-in configuration information about your Ruby installation, and you can get `irb` to load it by using `irb's -r` command-line flag and the name of the package:

```
$ irb --simple-prompt -rrbconfig
```

Now you can request information. Here's how to find out where the Ruby executable files (including `ruby` and `irb`) have been installed:

```
>> Config::CONFIG["bindir"]
```

To get additional installation information, you need to replace `bindir` in the `irb` command with other terms. But each time, you'll use the same basic formula: `Config::CONFIG["term"]`.

In each of the following sections, the section subtitle includes the term you need. Plug that term into the `irb` command, and you'll be shown the name of the directory.

### **1.2.1 The Ruby standard library subdirectory (*rubylibdir*)**

Inside `rubylibdir` (whatever that directory may be called on your system), you'll find program files written in Ruby. These files provide standard library facilities, which you can require from your own programs if you need the functionality they provide.

Here's a sampling of the files you'll find in this directory:

- *cgi.rb*—Tools to facilitate CGI programming
- *fileutils.rb*—Utilities for manipulating files easily from Ruby programs
- *tempfile.rb*—A mechanism for automating the creation of temporary files
- *tk.rb*—A programming interface to the Tk graphics library

Some of the standard libraries, such as the Tk graphics library (the last item on the previous list), span more than one file; you'll see a large number of files with names beginning with `tk`, as well as a whole `tk` subdirectory, all of which are part of the Ruby Tk library.

Browsing your `rubylibdir` will give you a good (if perhaps overwhelming, but only at first) sense of the many tasks for which Ruby provides programming facilities. Most programmers use only a subset of these capabilities, but even a subset of such a large collection of programming libraries gives you a lot to work with.

### **1.2.2 The C extensions directory (*archdir*)**

Usually located one level down from `rubylibdir`, `archdir` contains architecture-specific extensions and libraries. The files in this directory typically have names ending in `.so`, `.dll`, or `.bundle` (depending on your platform). These files are C-language extensions to Ruby; or,

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

more precisely, they're the binary, runtime-loadable files generated from Ruby's C-language extension code, compiled into binary form as part of the Ruby installation process.

Like the Ruby-language program files in `rubylibdir`, the files in `archdir` contain standard library components that you can load into your own programs. (Among others, you'll see the file for the `rbconfig` extension—the extension you're using with `irb` to uncover the directory names.) These files aren't human-readable, but the Ruby interpreter knows how to load them when asked to do so. From the perspective of the Ruby programmer, all standard libraries are equally useable, whether written in Ruby or written in C and compiled to binary format.

The files installed in `archdir` vary from one installation to another, depending on which extensions were compiled—which, in turn, depends on a mixture of what the person doing the compiling asked for and which extensions Ruby was able to compile.

### **1.2.3 The `site_ruby` (`sitedir`) and `vendor_ruby` (`vendordir`) directories**

Your Ruby installation includes a subdirectory called `site_ruby`. As its name suggests (albeit telegraphically), `site_ruby` is where you and/or your system administrator store third-party extensions and libraries. Some of these may be code you yourself write; others are tools you download from other people's sites and archives of Ruby libraries.

The `site_ruby` directory parallels the main Ruby installation directory, in the sense that it has its own subdirectories for Ruby-language and C-language extensions (`sitelibdir` and `sitearchdir`, respectively, in `Config` terms). When you require an extension, the Ruby interpreter checks for it in these subdirectories of `site_ruby` as well as in both the main `rubylibdir` and the main `archdir`.

Alongside `site_ruby` you'll find the directory `vendor_ruby`. Some third-party extensions install themselves here. The `vendor_ruby` directory is new as of Ruby 1.9, and standard practice as to which of the two areas gets which packages will probably develop over time.

### **1.2.4 The `gems` directory**

RubyGems is the standard way to package and distribute Ruby libraries. When you install *gems* (as the packages themselves are called), the unbundled library files land in the `gems` directory. You'll see more information about RubyGems, and how to install them, after we've looked at the basics of loading external program extensions and libraries. For now, we're just concerned with the presence of the `gems` directory. This directory is usually at the same level as `site_ruby`; if you've found `site_ruby`, look at what else is installed next to it. You won't see anything in the `gems` directory when you first install Ruby, but if you install some Ruby gems, you'll see them make their way into the directory.

Let's look now at the mechanics and semantics of how Ruby uses its own extensions as well as those you may write or install.

## 1.3 Ruby extensions and programming libraries

The first major point to take on board as you read this section is that it isn't a Ruby standard library reference. As explained in the introduction, this book doesn't aim to document the Ruby language; it aims to teach you the language and to confer Ruby citizenship upon you so that you can keep widening your horizons.

The purpose of this section, accordingly, is to show you how extensions work: how you get Ruby to run its extensions, the difference among techniques for doing so, and the extension architecture that lets you write your own extensions and libraries.

The extensions that ship with Ruby are usually referred to collectively as the *standard library*. Ruby comes with extensions for a wide variety of projects and tasks: database management, networking, specialized mathematics, XML processing, and many more. The exact makeup of the standard library usually changes, at least a little, with every new release of Ruby. But most of the more widely used libraries tend to stay, once they've proven their worth.

The key to using extensions and libraries is the `require` method, along with its near relation `load`. These methods allow you to load extensions at runtime, including extensions you write yourself. We'll look at them in their own right and then expand our scope to take in their use in loading built-in extensions.

### 1.3.1 Loading external files and extensions

Storing a program in a single file can be handy, but it starts to be a liability rather than an asset when you've got hundreds or thousands—or hundreds *of* thousands—of lines of code. Breaking your program into separate files then starts to make lots of sense. Ruby facilitates this process with the `require` and `load` methods. We'll start with `load`, which is the more simply engineered of the two.

#### Feature, extension, or library?

Things you load into your program at runtime get called several things. "Feature" is the most abstract, and is rarely heard outside of the specialized usage "requiring a feature" (i.e., with `require`). "Library" is more concrete, and more common. It connotes the actual code as well as the basic fact that a set of programming facilities exists and can be loaded. "Extension" can refer to any loadable add-on library, but is often used to mean a library for Ruby that has been written in the C programming language, rather than in Ruby. If you tell people you've written a Ruby extension, they'll probably assume you mean that it's in C.

To try the examples that follow, you'll need a program that's split over two files. The first file, `loaddemo.rb`, should contain the following Ruby code:

```
puts "This is the first (master) program file."  
load "loadee.rb"  
puts "And back again to the first file."
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

When it encounters the `load` method call, Ruby reads in the second file. That file, `loadee.rb`, should look like this:

```
puts "> This is the second file."
```

The two files should be in the same directory (presumably, your sample code directory).

When you run `loaddemo.rb` from the command line, you'll see this output:

```
This is the first (master) program file.
> This is the second file.
And back again to the first file.
```

The output gives you a trace of which lines, from which files, are being executed, and in what order.

The call to `load` in `loaddemo.rb` provides a filename, `loadee.rb`, as its argument:  
`load "loadee.rb"`

How does `loaddemo.rb` know where to find `loadee.rb`?

#### “LOAD”-ING A FILE IN THE DEFAULT LOAD PATH

The key to how `load` works is that the Ruby interpreter has knowledge of a *load path*: a list of directories in which it searches for files you ask it to load. You can see the names of these directories by examining the contents of the special global variable `$:` (dollar-colon). What you see depends on what platform you're on. A typical load-path inspection on Mac OS X looks like this:

```
$ ruby -e 'puts $:'      #A
/usr/local/lib/ruby/lib/ruby/site_ruby/1.9.0
/usr/local/lib/ruby/lib/ruby/site_ruby/1.9.0/i686-darwin9.1.0
/usr/local/lib/ruby/lib/ruby/site_ruby
/usr/local/lib/ruby/lib/ruby/vendor_ruby/1.9.0
/usr/local/lib/ruby/lib/ruby/vendor_ruby/1.9.0/i686-darwin9.1.0
/usr/local/lib/ruby/lib/ruby/vendor_ruby
/usr/local/lib/ruby/lib/ruby/1.9.0
/usr/local/lib/ruby/lib/ruby/1.9.0/i686-darwin9.1.0
.
```

**#A The `-e` switch lets you provide an inline script to ruby; see section 1.4.1.**

When you load a file, Ruby looks for it in each of these directories, in order. Note that the last directory is the current directory (represented by a dot). If the file isn't found anywhere else, it's sought in the current directory. When a program starts up, the current directory is the directory from which the program was started (not necessarily the same as the directory in which the startup program file resides, though it can be). It's possible to change the current directory in the course of running a program, in which case the significance of the dot changes.

You can navigate relative directories in your load commands with the conventional double-dot “directory up” symbol:

```
load "../extras.rb"
```

Note that if you change the current directory during a program run, relative directory references will change, too.

#### NOTE

Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

Keep in mind that `load` is a method, and it's executed at the point where Ruby encounters it in your file. Ruby doesn't search the whole file looking for load directives; it finds them when it finds them. This means you can load files whose names are determined dynamically; just put the call to `load` after the determination of the filename. You can even wrap a `load` call in a conditional statement, in which case it will be executed only if the condition is true.

You can also force `load` to find a file, regardless of the contents of the load path, by giving it the fully qualified path to the file:

```
load "/home/users/dblack/book/code/loadee.rb"
```

This is of course less portable than the use of the load path or relative directories, but it can be useful particularly if you have an absolute path stored as a string in a variable, and want to load the file it represents.

A call to `load` always loads the file you ask for, whether you've loaded it already or not. If a file changes between loadings, then anything in the new version of the file that rewrites or overrides anything in the original version takes priority. This can be useful, especially if you're in an `irb` session while you're modifying a file in an editor at the same time and want to examine the effect of your changes immediately.

The other file-loading method, `require`, also searches the directories that lie in the default load path. But `require` has some features that `load` doesn't have.

#### **"REQUIRE"-ING A FEATURE**

One major difference between `load` and `require` is that `require`, if called more than once with the same arguments, doesn't reload files it's already loaded. Ruby keeps track of which files you've required and doesn't duplicate the effort.

`require` is a little more abstract than `load`. Strictly speaking, you don't require a file; you require a feature. And typically, you do so without even specifying the extension on the filename. To see how this works, change this line in `loaddemo.rb`

```
load "loadee.rb"
```

to this:

```
require "loadee"
```

When you run `loaddemo.rb`, you get the same result as before, even though you haven't supplied the full name of the file you want loaded.

By viewing `loadee` as a "feature" rather than a file, `require` allows you to treat extensions written in Ruby the same way you treat extensions written in C—or, to put it another way, to treat files ending in `.rb` the same way as files ending in `.so`, `.dll`, or `.bundle`.

You can also feed a fully qualified path to `require`, as you can to `load`, and it will pull in the file/feature. And you can mix and match: the following syntax works, for example, even though it mixes the static path specification with the more abstract syntax of the "feature" at the end of the path:

```
require "/home/users/dblack/book/code/loadee"
```

Although `load` is useful, particularly when you want to load a file more than once, `require` is the day-to-day technique you'll use to load Ruby extensions and libraries—standard and

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

otherwise. Loading standard library features isn't any harder than loading `load_ee`. You just require what you want. Once you do, and of course depending on what the extension is, you'll have new classes and methods available to you. Here's a before-and-after example, in an irb session:

```
>> "David Black".scanf("%s%s")
NoMethodError: undefined method `scanf' for "David Black":String #1
>> require "scanf" #2
=> true
>> "David Black".scanf("%s%s")
=> ["David", "Black"] #3
```

## Cueballs in code and text

The first call to `scanf` fails with an error #1. After the `require` call, #2 however, and with no further programmer intervention, string objects like "David Black" respond to the `scanf` message. (In this example, #3 we're asking for two consecutive strings to be extracted from the original string, with whitespace as an implied separator.)

We'll conclude this chapter with an examination of the third of the three levels mapped out at the beginning of the chapter: the command-line tools that ship with Ruby.

### 1.4 Out-of-the-box Ruby tools and applications

When you install Ruby, you get a handful of important command-line tools, which are installed in whatever directory is configured as `bindir`—usually `/usr/local/bin`, `/usr/bin`, or the `/opt` equivalents. (You can `require "rbconfig"` and examine `Config::CONFIG["bindir"]` to check.) These tools are: `erb`, `gem`, `irb`, `rake`, `rdoc`, `ri`, `ruby` (the interpreter itself), and `testrb`.

In this section we'll look at each of these tools, with the exception of `testrb`. We'll start with `ruby`, looking at some of its command-line switches. From there, we'll go in turn through the other six.

#### 1.4.1 Interpreter command-line switches

When you start the Ruby interpreter from the command line, you can provide not only the name of a program file but also one or more command-line switches. The switches you choose instruct the interpreter to behave in particular ways and/or take particular actions.

Ruby has more than 20 command-line switches. Some of them are used rarely; others are used every day by many Ruby programmers. You've already seen examples of a few of them. Here we'll look at those in more detail, and at several more of the most commonly used ones.

These switches are summarized in table 1.2 and then explained separately.

Table 1.2 Summary of commonly used Ruby command-line switches

Switch	Description	Example of usage
--------	-------------	------------------

Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

Switch	Description	Example of usage
-c	Check the syntax of a program file without executing the program	<code>ruby -c c2f.rb</code>
-w	Give warning messages during program execution	<code>ruby -w c2f.rb</code>
-e	Execute the code provided in quotation marks on the command line	<code>ruby -e 'puts "Code demo!"'</code>
-v	Show Ruby version information, and execute the program in verbose mode	<code>ruby -v</code>
-l	Line mode: print a newline after every line, if not otherwise present	<code>ruby -le 'print "Will jump down!"'</code>
-r <i>name</i>	Require the named extension	<code>ruby -rprofile</code>
--version	Show Ruby version information	<code>ruby --version</code>

**In the following heading, make the text in the parentheses regular font, not small caps, so the letter after the dash is lowercase**

#### **CHECK SYNTAX (-c)**

The `-c` switch tells Ruby to check the code in one or more files for syntactical accuracy without executing the code. It's usually used in conjunction with the `-w` flag.

**In the following heading, make the text in the parentheses regular font, not small caps, so the letter after the dash is lowercase**

#### **TURN ON WARNINGS (-w)**

Running your program with `-w` causes the interpreter to run in warning mode. This means you see more warnings than you otherwise would printed to the screen, drawing your attention to places in your program that, although not syntax errors, are stylistically or logically suspect. It's Ruby's way of saying, "What you've done is syntactically correct, but it's weird. Are you sure you meant to do that?" (Even without this switch, Ruby issues certain warnings, but fewer than it does in full warning mode.)

**In the following heading, make the text in the parentheses regular font, not small caps, so the letter after the dash is lowercase**

Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

**EXECUTE LITERAL SCRIPT (-E)**

The `-e` switch tells the interpreter that the command line includes Ruby code, in quotation marks, and that it should execute that actual code rather than executing the code contained in a file. This can be handy for quick scripting jobs where entering your code into a file and running ruby on the file may not be worth the trouble.

For example, let's say you want to see your name printed out backwards. Here's how you can do this quickly, in one command-line command, using the execute switch:

```
$ ruby -e 'puts "David A. Black".reverse'
kcalB .A divaD
```

What lies inside the single quotation marks is an entire (although short) Ruby program. If you want to feed a program with more than one line to the `-e` switch, you can use literal line breaks inside the mini-program:

```
$ ruby -e 'print "Enter a name: "
puts gets.reverse'
Enter a name: David A. Black
```

```
kcalB .A divaD
```

Or, you can separate the lines with semicolons:

```
$ ruby -e 'print "Enter a name: "; print gets.reverse'
```

**NOTE**

Why is there a blank line between the program code and the output in the two-line reverse examples? Because the line you enter on the keyboard ends with a newline character—so when you reverse the input, the new string starts with a newline! Ruby, as always, takes you literally when you ask it to manipulate and print data.

**In the following heading, make the text in the parentheses regular font, not small caps, so the letter after the dash is lowercase**

**RUN IN LINE MODE (-L)**

The `-l` switch produces the effect that every string output by the program is placed on a line of its own, even if it normally would not be. Usually this means that lines that are output using `print`, rather than `puts`, and which therefore don't automatically end with a newline character, now end with a newline.

We made use of the `print` versus `puts` distinction to ensure that our temperature converter programs didn't insert extra newlines in the middle of their output. (See Section 1.1.5.) You can use the `-l` switch to reverse the effect; it causes even printed lines to appear on a line of their own. Here's the difference:

```
$ ruby c2f-2.rb
The result is 212.
$ ruby -l c2f-2.rb           #A
The result is
212
.
```

**#A -l ensures every output line on a separate line**

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

The result with `-l` is, in this case, exactly what we don't want. But the example illustrates the effect of the switch. Cases where you do want the effect of automatic newlines are typically cases where you're not sure whether the values being printed out do or don't end with newlines, and you want to force uniformity in the output.

The `-l` switch isn't commonly used or seen, largely because of the availability of `puts` to ensure similar behavior. But it's good to know it's there and to be able to recognize it.

**In the following heading, make the text in the parentheses regular font, not small caps, so the letters after the dash are lowercase**

#### **REQUIRE NAMED FILE OR EXTENSION (-RNAME)**

The `-r` switch calls `require` on its argument; `ruby -rscanf` will `require` `scanf` when the interpreter starts up. You can put more than one `-r` switch on a single command line.

**In the following heading, make the text in the parentheses regular font, not small caps, so the letters after the dash are lowercase**

#### **RUN IN VERBOSE MODE (-v, --VERBOSE)**

Running with `-v` does two things: it prints out information about the version of Ruby you're using, and then it turns on the same warning mechanism as the `-w` flag. The most common use of `-v` is to find out the Ruby version number:

```
$ ruby -v
ruby 1.9.0 (2007-12-25 revision 14709) [i686-darwin8.11.1]
```

In this case, we're using Ruby 1.9.0, released on Christmas Day, 2007, and compiled for an i686-based machine running Mac OS X. Because there's no program or code to run, Ruby exits as soon as it has printed the version number.

**In the following heading, make the text in the parentheses regular font, not small caps, so the letters after the dash are lowercase. Also, it's a double-dash in this case (--version).**

#### **PRINT RUBY VERSION (--VERSION)**

This flag causes Ruby to print a version information string and then exit. It doesn't execute any code, even if you provide code or a filename. You'll recall that `-v` also prints version information and then runs your code (if any) in verbose mode. You might say that `-v` is slyly standing for both *version* and *verbose*, whereas `--version` is just *version*.

**In the following heading, make the text in the parentheses regular font, not small caps, so the letters after the dash are lowercase**

**PRINT SOME HELP INFORMATION (-H, --HELP)**

These switches give you a table listing all the command-line switches available to you, and summarizing what they do.

In addition to using single switches, you can also combine two or more in a single invocation of Ruby.

**COMBINING SWITCHES**

You've already seen the `cw` combination, which checks the syntax of the file without executing it, while also giving you warnings:

```
$ ruby -cw filename
```

Another combination of switches you'll often see is `-v` and `-e`, which shows you the version of Ruby you're running and then runs the code provided in quotation marks. You'll see this combination a lot in discussions of Ruby, on mailing lists and elsewhere; people use it to demonstrate how the same code might work differently in different versions of Ruby. For example, if you want to show clearly that a string method called `start_with?` wasn't present in Ruby 1.8.6 but is present in Ruby 1.9.0, you can run a sample program using first one version of Ruby and then the other:

```
$ ruby186 -ve "puts 'abc'.start_with?('a')"
ruby 1.8.6 (2007-03-13 patchlevel 0) [i686-darwin8.10.1]
-e:1: undefined method `start_with?' for "abc":String (NoMethodError) #1

$ ruby -ve "puts 'abc'.start_with?('a')"
ruby 1.9.0 (2007-12-25 revision 14709) [i686-darwin8.11.1]
true #2
```

## Cueballs in code and text

(Of course, you must have both versions of Ruby installed on your system.) The undefined method `'start_with?'` message on the first run (the one using version 1.8.6) #1 means that you've tried to perform a nonexistent named operation. When you run the same Ruby snippet using Ruby 1.9.0, however, it works: Ruby prints `true` #2. This is a convenient way to share information and formulate questions about changes in Ruby's behavior from one release to another.

**TIP:**

You can feed Ruby the switches separately, like this:

## Following code line is part of the Tip.

```
$ ruby -c -w
or
```

## Following code line is part of the Tip.

Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

```
$ ruby -v -e "puts 'abc'.start_with?('a')"
```

But it's common to type them together, as in the examples in the main text.

At this point, we'll go back and look more closely at the interactive Ruby interpreter, `irb`. You may have looked at this section already, when it was mentioned near the beginning of the chapter. If not, you can take this opportunity to learn more about this exceptionally useful Ruby tool.

### 1.4.2 A closer look at interactive Ruby interpretation with `irb`

As you've seen, `irb` is an interactive Ruby interpreter—which means that instead of processing a file, it processes what you type in during a session. `irb` is a great tool for testing Ruby code, and a great tool for learning Ruby.

To start an `irb` session, you use the command `irb`. `irb` prints out its prompt:

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

As you've seen, you can also use the `--simple-prompt` option to keep `irb`'s output shorter:

```
$ irb --simple-prompt
>>
```

Once `irb` starts, you can enter Ruby commands. You can even run a one-shot version of the Celsius-to-Fahrenheit conversion program. As you'll see in this example, `irb` behaves like a pocket calculator: it evaluates whatever you type in and prints the result. You don't have to use a `print` or `puts` command:

```
>> 100 * 9 / 5 + 32
=> 212
```

To find out how many minutes there are in a year (if you don't have a CD of the relevant hit song from the musical *Rent* handy), type in the appropriate multiplication expression:

```
>> 365 * 24 * 60
=> 525600
```

`irb` will also, of course, process any Ruby instructions you enter. For example, if you want to assign the day, hour, and minute counts to variables, and then multiply those variables, you can do that in `irb`:

```
>> days = 365
=> 365
>> hours = 24
=> 24
>> minutes = 60
=> 60
>> days * hours * minutes
=> 525600
```

The last calculation is what you'd expect. But look at the first three lines of entry. When you type `days = 365`, `irb` responds by printing 365. Why?

The expression `days = 365` is an assignment expression: you're assigning the value 365 to a variable called `days`. The main business of an assignment expression is to assign, so that you can use the variable later. But assignment expressions themselves—such as the entire line `days = 365`—have a value. The value of an assignment expression is its right-

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

hand side. When irb sees any expression, it prints out the value of that expression. So, when irb sees `days = 365`, it prints out 365. This may seem like overzealous printing, but it comes with the territory; it's the same behavior that lets you type `2 + 2` into irb and see the result without having to use an explicit `print` statement.

Similarly, even a call to the `puts` method has a return value: namely, `nil`. If you type a `puts` statement in irb, irb will obediently execute it, and will also print out the return value of `puts`:

```
$ irb --simple-prompt
>> puts "Hello"
Hello
=> nil
```

There's a way to get irb not to be quite so talkative: the `--noecho` flag. Here's how it works:

```
$ irb --simple-prompt --noecho
>> 2 + 2
>> puts "Hi"
Hi
```

Thanks to `--noecho`, the addition expression doesn't report back its result. The `puts` command does get executed (so you see "Hi"), but the return value of `puts` (`nil`) is suppressed.

### TIP

It's possible to get stuck in a loop in irb, or for the session to feel like it's not responding (which often means you've typed an opening quotation mark but not a closing one, or something along those lines). How you get control back is somewhat system-dependent. On most systems, `Ctrl-c` will do the trick. On others, you might need to use `Ctrl-z`. It's best to apply whatever general program-interrupting information you have about your system directly to irb. Of course, if things get really frozen you can go to your process or task management tools and kill the process.

To exit from irb normally, you can just type `exit`.

Occasionally, irb may blow up on you (that is, hit a fatal error and terminate itself). Most of the time, though, it catches its own errors and lets you continue.

Once you get the hang of irb's approach to printing out the value of everything, and how to shut it up if you want to, you'll find it an immensely useful tool (and toy).

Ruby's source code is marked up in such a way as to provide for automatic generation of documentation; and the tools needed to interpret and display that documentation are `ri` and `RDoc`, which we'll look at now.

### 1.4.5 `ri` and `RDoc`

`ri` (Ruby Index) and `RDoc` (Ruby Documentation) are a closely related pair of tools for providing documentation about Ruby programs. `ri` is a command-line tool; the `RDoc` system

Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

includes the command-line tool `rdoc`. `ri` and `rdoc` are standalone programs; you run them from the command line. You can also use the facilities they provide from within your Ruby programs.

RDoc is a documentation system. If you put comments in your program files (Ruby or C) in the prescribed RDoc format, `rdoc` scans your files, extracts the comments, organizes them intelligently (indexed according to what they comment on), and creates nicely formatted documentation from them. You can see RDoc markup in many of the source files, in both Ruby and C, in the Ruby source tree, and in many of the Ruby files in the Ruby installation.

### RDoc and the SimpleMarkup system

RDoc uses a documentation format called SimpleMarkup, which provides a basic syntax for formatting comments in source code so that the RDoc processor can make sense of them and output them in a useful format (HTML, plain text, etc.). SimpleMarkup uses XML-style tags and other markup devices to indicate for formatting and typesetting instructions; for example, `<em>word</em>` indicates that “word” should be in an emphasis font (such as italics), and `+word+` means that “word” should be in a code font (usually monospace). You can also construct lists of various kinds (numbered, labeled, bullet) using SimpleMarkup.

The RDoc system itself reads one or more files annotated with SimpleMarkup instructions, and generates output in a given format. Plain text output serves as the basis for `ri`. HTML output from RDoc is often used in online documentation. RDoc also has intelligence about where in the document comments are placed; for example, comment lines right before a method definition are considered that method’s documentation, and positioned accordingly in the output.

You can mark up your own source files with SimpleMarkup and use RDoc to generate documentation. For more on the specifics of these tools, you’ll find the documentation in the files themselves very informative. See the `rdoc` subdirectory of your Ruby installation, especially the file `rdoc/markup/simple_markup.rb`. Try adding some SimpleMarkup instructions to a Ruby source file; then run the command `rdoc filename` on the file. RDoc will create a subdirectory called `doc`, containing documentation for your file in HTML format. It’s that easy!

`ri` dovetails with RDoc: it gives you a way to view the information that RDoc has extracted and organized. Specifically (although not exclusively, if you customize it), `ri` is configured to display the RDoc information from the Ruby source files. Thus on any system that has Ruby fully installed, you can get detailed information about Ruby with a simple command-line invocation of `ri`.

For example, here’s how you would request information about the `upcase` method of string objects:

Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

```
$ ri String#upcase
```

And here's what you get back:

```
----- String#upcase
str.upcase => new_str
-----
Returns a copy of _str_ with all lowercase letters replaced with
their uppercase counterparts. The operation is locale
insensitive---only characters 'a' to 'z' are affected. Note:
case replacement is effective only in ASCII region.

"hello".upcase #=> "HELLO"
```

The hash mark (#) between `String` and `upcase` in the `ri` command indicates that you're looking for an instance method, as distinct from a class method. In the case of a class method, you'd use the separator `::` instead of `#`. We haven't looked at the class method/instance method distinction yet, and you don't have to concern yourself with it now. The main point for the moment is that you have lots of documentation at your disposal from the command line.

### TIP: GETTING OUT OF RI

By default, `ri` runs its output through a pager (such as `more` on Unix). It may pause at the end of output, waiting for you to hit the spacebar or some other key to show the next screenful of information or to exit entirely if all the information has been shown. Exactly what you have to press in this case varies from one operating system, and one pager, to another. Spacebar, enter, control-c, control-d, and control-z are all good bets. If you want `ri` to write the output without filtering it through a pager, you can use the `-T` commandline switch (`ri -T topic`).

Next among the Ruby command-line tools is `rake`.

### 1.4.6 The rake task-management utility

As its name suggests (it comes from "Ruby make"), `rake` is a make-inspired task-management utility. It's written by Jim Weirich. Like `make`, `rake` reads and executes tasks defined in a file—a `Rakefile`. Unlike `make`, however, `rake` uses Ruby syntax to define its tasks.

Listing 1.5 shows a `Rakefile`. If you save the listing as a file called `Rakefile`, you can then issue this command at the command line:

```
$ rake admin:clean_tmp
```

`Rake` executes the `clean_tmp` tasks defined inside the `admin` namespace. #1

#### Listing 1.5 Sample `Rakefile` defining a `clean_tmp` tasks inside the `admin` namespace

```
namespace :admin do #1
  desc "Interactively delete all files in /tmp" #2
  task :clean_tmp do
    Dir["/tmp/*"].each do |f| #3
      next unless File.file?(f) #4
      print "Delete #{f}? " #5
    end
  end
end
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

```

        answer = $stdin.gets
        case answer
        when /^y/
            File.unlink(f)                #6
        when /^q/
            break                          #7
        end
    end
end
end
end

```

The rake task defined here uses several Ruby techniques that you haven't seen yet. The basic algorithm, however, is pretty simple. Loop through each directory entry in the /tmp directory #3. Skip the current loop iteration unless this entry is a file #4. (Hidden files aren't deleted either, because the directory listing operation doesn't include them.) Prompt for the deletion of the file #5. If the user types `y` (or anything beginning with `y`), the file is deleted #6. If the user types `q`, break out of the loop; the task stops #7. The main programming logic comes from the looping through the list of directory entries (see Note, below) and from the case statement, a conditional execution structure. (You'll see both of these techniques in detail later in the book.)

#### NOTE: USING EACH TO LOOP THROUGH A COLLECTION

The expression `Dir["/tmp/*"].each do |f|` is a call to the `each` method of the array of all the directory entry names. The entire block of code starting with `do` and ending with `end` (the `end` that lines up with `Dir` in the indentation) gets executed once for each item in the array. Each time through, the current item is bound to the parameter `f`; that's the significance of the `|f|` part. You'll see `each` several times in the coming chapters, and we'll examine it in detail when we look at *iterators* (methods that automatically traverse collections) in chapter 9.

The `desc` command above the task definition #2 provides a description of the task. This comes in handy not only when you're perusing the file but also if you want to see all the tasks that rake can execute at a given time. If you're in the directory containing the Rakefile in listing 1.5 and you give this command

```
$ rake --tasks
```

you see a listing of all defined tasks:

```
$ rake --tasks
(in /Users/ruby/hacking)
rake admin:clean_tmp # Interactively delete all files in /tmp
```

You can use any names you want for your rake namespaces and tasks. You don't even need a namespace; you can define a task at the top-level namespace

```
task :clean_tmp do
  # etc.
end
```

and then invoke it using the simple name:

```
$ rake clean_tmp
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

However, namespacing your tasks is a good idea, particularly if and when the number of tasks you define grows significantly. You can namespace to any depth; this structure, for example, is legitimate:

```
namespace :admin do
  namespace :clean do
    task :tmp do
      # etc.
    end
  end
end
```

The task defined here is invoked like this:

```
$ rake admin:clean:tmp
```

As the directory-cleaning example shows, rake tasks don't have to be confined to actions related to Ruby programming. With rake, you get the whole Ruby language at your disposal, for the purpose of writing whatever tasks you need.

The next tool on the tour is the `gem` command, which makes installation of third-party Ruby packages very easy.

### 1.4.7 Installing packages with the `gem` command

As of version 1.9, Ruby ships with the RubyGems package-management utility. RubyGems includes facilities for packaging and installing Ruby libraries and applications. We're not going to cover gem creation here, but we'll look at gem installation and usage.

#### **WARNING: RUBYGEMS IN RUBY 1.9**

Ruby gems—the gem-packaged libraries and applications themselves—have suffered some transition pains between Ruby 1.8 and Ruby 1.9. A major problem is that Ruby's C API has changed, so gems that rely on compiling C files aren't likely to work at all without changes, and not all of them have been update. Also, features loaded with `require` in 1.8 may not be found in 1.9 if they have been moved, removed, or renamed.

For these reasons, the examples here are actually based on installing and loading gems on a Ruby 1.8 installation.

Stay tuned, though; things will smooth out with respect to gems and 1.9. If you have problems installing a specific gem, try searching for information; it's not unlikely that someone else will have encountered the same problem and posted a solution.

Installing a Ruby gem can be, and usually is, as easy as issuing a simple `install` command.

```
$ gem install rupert
```

Such a command gives you output something like the following (depending on which gems you already have installed and which dependencies have to be met by installing new gems):

```
Successfully installed hoe-1.7.0
Successfully installed color-1.4.0
Successfully installed transaction-simple-1.4.0
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

```
Successfully installed pdf-writer-1.1.8
Successfully installed ruport-1.6.1
5 gems installed
```

These status reports will be followed by several lines indicating that ri and RDoc documentation for the various gems is being installed. (The installation of the documentation involves processing the gem source files through RDoc, so be patient; this is often the longest phase of gem installation.)

During the gem installation process, gem downloads a number of gem files from RubyForge. Those files, which are in .gem format, are saved in the cache subdirectory of your gems directory.

You can also install a gem from a gem file residing locally on your hard disk or other storage medium. Give the name of the file to the installer:

```
$ gem install /home/me/mygems/ruport-1.4.0.gem
```

If you name a gem without the entire filename (for example, ruport), gem looks for it in the current directory and in the local cache maintained by the RubyGems system. Moreover, local installations still search remotely for dependencies, unless you provide the `-l` (local) command-line flag to the gem command; that flag restricts all operations to the local domain. If you want only remote gems installed, including dependencies, then you can use the `-r` (remote) flag. In most cases, the simple gem `gemname` command will give you what you need.

Once you've got a gem installed, you can use it, although you need to know a trick or two about how to load a gem as opposed to a non-gem library.

### LOADING AND USING GEMS

The key to loading a gem is to get that gem's directory into the load path. Then you can require it.

Here's an irb session that shows you how to load a gem, along with some examination of the load path (`$:`) along the way. (Long file paths have been abbreviated with `'...'` in the output listed below.)

```
>> require "rubygems"          #1
=> true
>> require "active_support"    #2
=> true
>> puts $:
.../gems/activesupport-2.0.2/lib/active_support/vendor/xml-simple-1.0.11
.../gems/activesupport-2.0.2/lib/active_support/vendor/builder-2.1.2
.../gems/1.8/gems/activesupport-2.0.2/lib
.../gems/1.8/gems/activesupport-2.0.2/bin
.../gems/1.8/gems/activesupport-2.0.2/lib
/usr/local/lib/ruby/site_ruby/1.8
/usr/local/lib/ruby/site_ruby/1.8/i686-linux
/usr/local/lib/ruby/site_ruby
/usr/local/lib/ruby/1.8
/usr/local/lib/ruby/1.8/i686-linux
.
=> nil
>> 3.days.ago
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

```
=> Sat Feb 23 09:33:19 -0500 2008
```

First, you have to load the `rubygems` library #1. (This is an artifact of Ruby 1.8. In 1.9, the library is loaded automatically.) Then, you can require a feature that's stored as a gem #2. Once you do this, the load path is modified to include directories related to ActiveSupport, including two subdirectories that house other gems (`xml-simple` and `builder`). And ActiveSupport is now loaded, as you can see from the time manipulation example that uses it #3.

You can insert gem-related directories in the load path in a slightly less transparent way, using the `gem` method. (Note that this method isn't the same as the command-line tool called `gem`.) If you call `gem "activesupport"`, for example, the two principal search directories for ActiveSupport are placed at the front of the load path:

```
>> require "rubygems"
=> true
>> gem "activesupport"
=> true
>> puts $:
/usr/local/lib/ruby/gems/1.8/gems/activesupport-2.0.2/bin
/usr/local/lib/ruby/gems/1.8/gems/activesupport-2.0.2/lib
/usr/local/lib/ruby/site_ruby/1.8
/usr/local/lib/ruby/site_ruby/1.8/i686-linux
/usr/local/lib/ruby/site_ruby
/usr/local/lib/ruby/1.8
/usr/local/lib/ruby/1.8/i686-linux
.
```

(This particular gem name is indeed spelled without an underscore, even though the package name that you see in the `require` examples is `active_support`.) Having adjusted the load path with the `gem` command, you still have to use `require` to get the feature you want loaded. Here's a continuation of the last irb transcript:

```
>> 3.days.ago
NoMethodError: undefined method 'days' for 3:Fixnum
from (irb):4
>> require "active_support"
=> true
>> 3.days.ago
=> Sat Feb 23 09:45:45 -0500 2008
```

(You need to include the underscore for the argument to `require`, for reasons having to do specifically with how ActiveSupport's load files are named and laid out.) The `gem` method ensures that a given path, or paths, come first in the load path, which can be helpful if you're worried that `require` might get confused. If you happen to have a file called `activesupport.rb` in your working directory, for example, `require` will load it in preference to the gem; it doesn't add the directories to the front of the load path until after it's loaded the feature you've requested. The `gem` method lets you manipulate the load path first and `require` features after.

Moreover, `gem` lets you target specific versions of gems. It takes a second argument: a version number. It's not uncommon to have more than one version of a given gem installed on your system, and there may be reasons for wanting to load a version other than the

Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

latest one. You can do this by specifying a version—or a minimum or maximum version. In this example, we're asking for a version less than or equal to 1.4.4. The inspection of \$: shows that we've got it, and we can then require it:

```
>> require "rubygems"
=> true
>> gem "activesupport", "<= 1.4.4"
=> true
>> puts $:
/usr/local/lib/ruby/gems/1.8/gems/activesupport-1.4.4/bin
/usr/local/lib/ruby/gems/1.8/gems/activesupport-1.4.4/lib
/usr/local/lib/ruby/site_ruby/1.8
/usr/local/lib/ruby/site_ruby/1.8/i686-linux
/usr/local/lib/ruby/site_ruby
/usr/local/lib/ruby/1.8
/usr/local/lib/ruby/1.8/i686-linux
.
=> nil
>> require "active_support"
=> true
```

The require command doesn't know about gems as such; it just knows that the 1.4.4 branch of ActiveSupport now lies on the load path.

We've got one more stop on the tour of out-of-the-box command-line tools for Ruby: ERb (Embedded Ruby).

### 1.4.8 ERb

ERb is a templating system that lets you include executable Ruby code along with static text in a file. The file is the input to ERb; the output is a stream of characters comprising the static text from the original file interpolated with the results of evaluating the code. You can also insert code into an ERb template for execution without having it inserted into the output stream.

ERb is like an automated Mad Lib. It reads along, word for word; then, at a certain point (when it sees the Ruby code embedded in the document), it sees that it has to fill in a blank, which it does by executing the Ruby code.

You need to know only two things to prepare an ERb document:

- If you want some Ruby code executed, enclose it between <% and %>.
- If you want the result of the code execution to be printed as part of the output, enclose the code between <%= and %>.

ERb figures out what to do when it hits <% or <%=.

Here's an example. Save the code from listing 1.6 in your rubycode directory as erbdemo.rb.

#### Listing 1.6 Demonstration of ERb (erbdemo.rb)

```
<% page_title = "Demonstration of ERb" %>
<% salutation = "Dear programmer," %>
<html>
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

```

<head>
<title><%= page_title %></title>
</head>
<body>
<p><%= salutation %></p>
<p>This is a brief demonstration of how ERb fills out a template.</p>
</body>
</html>

```

Now, run the program using the command-line utility `erb` instead of `ruby`:

```
$ erb erbdemo.rb
```

```

<html>
<head>
<title>Demonstration of ERb</title> #5
</head>
<body>
<p>Dear programmer,</p>
<p>This is a brief demonstration of how ERb fills out a template.</p>
</body>
</html>

```

The output of the program run is just what you'd expect, given the rules for how ERb reads and interprets its input. The first two lines of the program are interpreted as Ruby instructions (that is, the parts inside the `<%...%>` markers; the markers themselves are ignored). Once those two lines have been read, you have two variables to work with: `page_title` and `salutation`. The HTML markup instruction `<html>` is read in literally and printed out literally, with no change. That's the first line of output (except for two blank lines; `erb` gave you a blank line for each of those `<%...%>` lines). The `<head>` tag also comes through in the output just as it appeared in the input.

In the `<title>` tag, you see some Ruby code inside a `<%=...%>` delimiter pair. These are the delimiters you use when you want the result of evaluating the code to be inserted into the ERb output. The Ruby code, in this case, is the single variable `page_title`, and the value of that variable is the string "Demonstration of ERb". (You know this because you assigned that value to the `title` variable on the first line.) At this point in the output, ERb fills in the perceived blank with "Demonstration of ERb".

ERb is used heavily in Web application templating and is the default templating system of the Ruby on Rails development framework. It's also handy for semi-automating the creation of data. If, for example, you wanted a file containing the names of the cards in a deck, you could create an ERb file like this:

```

<% ["hearts", "diamonds", "spades", "clubs"].each do |suit| %>
  <% ["2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A"].each do |rank|
    %>
    <%= rank %> of <%= suit %> <% end %> <% end %>
  <% end %>
<% end %>

```

(The two `end` instructions are run together on the last line so as to minimize the insertion of extra blank lines.) You can achieve similar results, at least in some cases, by printing out the strings. But ERb comes in handy when you want a program to read from a file and you need a convenient way to generate lines without having to write them out.

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

At this point, we're on the brink of moving away from command-line tools as such. Most program-to-program ERb activity uses not the `erb` command but the ERb API, which allows you to process strings (including file input) through ERb from inside a Ruby program. You do this by loading the ERb library and creating a new ERB (with a capital B!) object:

```
>> require 'erb'  
=> true  
>> card_string = ERB.new(File.read("cards.erb")).run
```

```
2 of hearts  
3 of hearts  
4 of hearts  
etc.
```

The vista of the standard library is impressive, but we're finished with our current business—the `bin/` directory—and we'll move next to a close study of the core language.

## 1.5 Summary

In this chapter, we've walked through important foundational Ruby material and facilities. You've learned some key terminology, including the difference between Ruby (the programming language) and `ruby` (the name of the Ruby interpreter program). You've completed (in miniature, but still from start to finish) the process of writing a Ruby program, saving it in a file, checking it for syntax errors, and running it. You've gotten a taste of how to do keyboard input in Ruby as well as file input and output. You've also learned how to pull in one program file from another with `require` and `load`.

Section 1.2 provided a tour of the Ruby installation. In section 1.3, we looked at how Ruby handles loading extensions and libraries, from the humblest second file you happen to write to the mightiest C library. Section 1.4 introduced you to the command-line tools that Ruby ships with.

You now have a good blueprint of how Ruby works and what tools the Ruby programming environment provides, and you've seen and practiced some important Ruby techniques. You're now prepared to start exploring Ruby systematically.