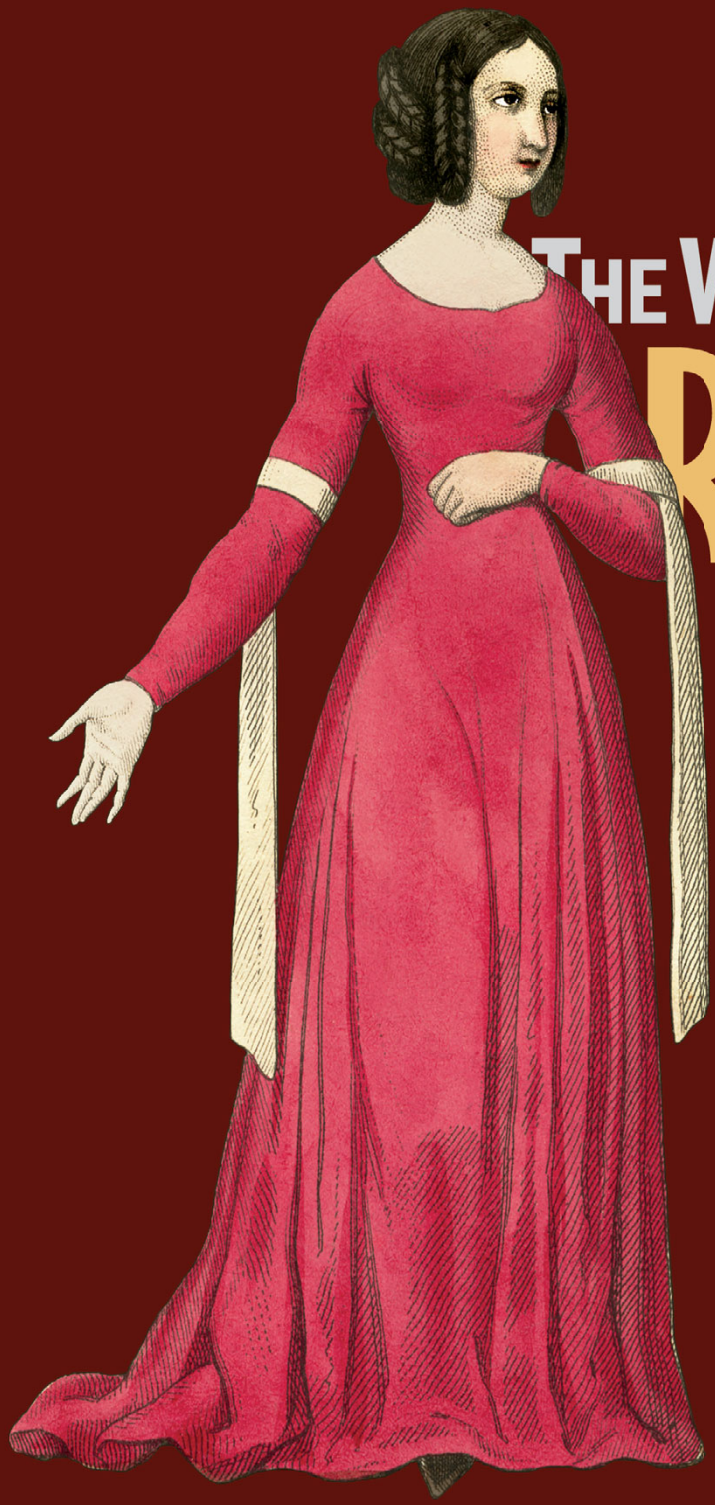
SAMPLE CHAPTER



THE WELL-GROUNDED Rubyist

SECOND EDITION

David A. Black

MANNING

The Well-Grounded Rubyist, Second Edition

by David A. Black

Chapter 12

brief contents

12 File and I/O operations

This chapter covers

- Keyboard input and screen output
- The `IO` and `File` classes
- Standard library file facilities, including `FileUtils` and `Pathname`
- The `StringIO` and `open-uri` library features

As you'll see once you dive in, Ruby keeps even file and I/O operations object-oriented. Input and output streams, like the standard input stream or, for that matter, any file handle, are objects. Some I/O-related commands are more procedural: `puts`, for example, or the `system` method that lets you execute a system command. But `puts` is only procedural when it's operating on the standard output stream. When you `puts` a line to a file, you explicitly send the message "puts" to a `File` object.

The memory space of a Ruby program is a kind of idealized space, where objects come into existence and talk to each other. Given the fact that I/O and system command execution involve stepping outside this idealized space, Ruby does a lot to keep objects in the mix.

You'll see more discussion of standard library (as opposed to core) packages in this chapter than anywhere else in the book. That's because the file-handling

facilities in the standard library—highlighted by the `FileUtils`, `Pathname`, and `StringIO` packages—are so powerful and so versatile that they've achieved a kind of quasi-core status. The odds are that if you do any kind of file-intensive Ruby programming, you'll get to the point where you load those packages almost without thinking about it.

12.1 How Ruby's I/O system is put together

The `IO` class handles all input and output streams either by itself or via its descendant classes, particularly `File`. To a large extent, `IO`'s API consists of wrappers around system library calls, with some enhancements and modifications. The more familiar you are with the C standard library, the more at home you'll feel with methods like `seek`, `getc`, and `eof?`. Likewise, if you've used another high-level language that also has a fairly close-fitting wrapper API around those library methods, you'll recognize their equivalents in Ruby. But even if you're not a systems or C programmer, you'll get the hang of it quickly.

12.1.1 The IO class

`IO` objects represent readable and/or writable connections to disk files, keyboards, screens, and other devices. You treat an `IO` object like any other object: you send it messages, and it executes methods and returns the results.

When a Ruby program starts up, it's aware of the standard input, output, and error streams. All three are encapsulated in instances of `IO`. You can use them to get a sense of how a simple `IO` object works:

```
>> STDERR.class
=> IO
>> STDERR.puts("Problem!")
Problem!
=> nil
>> STDERR.write("Problem!\n")
Problem!
=> 9
```

❶ ❷ ❸

The constants `STDERR`, `STDIN`, and `STDOUT` (all of which will be covered in detail in section 12.1.3) are automatically set when the program starts. `STDERR` is an `IO` object ❶. If an `IO` object is open for writing (which `STDERR` is, because the whole point is to output status and error messages to it), you can call `puts` on it, and whatever you `puts` will be written to that `IO` object's output stream ❷. In the case of `STDERR`—at least, in the default startup situation—that's a fancy way of saying that it will be written to the screen.

In addition to `puts`, `IO` objects have the `print` method and a `write` method. If you `write` to an `IO` object, there's no automatic newline output (`write` is like `print` rather than `puts` in that respect), and the return value is the number of bytes written ❸.

`IO` is a Ruby class, and as a class it's entitled to mix in modules. And so it does. In particular, `IO` objects are enumerable.

12.1.2 IO objects as enumerables

An enumerable, as you know, must have an `each` method so that it can iterate. `IO` objects iterate based on the global input record separator, which, as you saw in connection with strings and their `each_line` method in section 10.7, is stored in the global variable `$/`.

In the following examples, Ruby's output is indicated by **bold** type; regular type indicates keyboard input. The code performs an iteration on `STDIN`, the standard input stream. (We'll look more closely at `STDIN` and friends in the next section.) At first, `STDIN` treats the newline character as the signal that one iteration has finished; it thus prints each line as you enter it:

```
>> STDIN.each {|line| p line }
This is line 1
"This is line 1\n"
This is line 2
"This is line 2\n"
All separated by $/, which is a newline character
"All separated by $/, which is a newline character\n"
```

But if you change the value of `$/`, `STDIN`'s idea of what constitutes an iteration also changes. Terminate the first iteration with Ctrl-d (or Ctrl-c, if necessary!), and try this example:

```
>> $/ = "NEXT"
=> "NEXT"
>> STDIN.each {|line| p line}
First line
NEXT
"First line\nNEXT"
Next line
where "line" really means
until we see... NEXT
"\nNext line\nwhere \"line\" really means\nuntil we see... NEXT"
```

Here, Ruby accepts keyboard input until it hits the string `"NEXT"`, at which point it considers the entry of the record to be complete.

So `$/` determines an `IO` object's sense of "each." And because `IO` objects are enumerable, you can perform the usual enumerable operations on them. (You can assume that `$/` has been returned to its original value in these examples.) The `^D` notation indicates that the typist entered Ctrl-d at that point:

```
>> STDIN.select {|line| line =~ /\A[A-Z]/ }
We're only interested in
lines that begin with
Uppercase letters
^D
=> ["We're only interested in\n", "Uppercase letters\n"]
>> STDIN.map {|line| line.reverse }
senil esehT
terces a niatnoc
.egassem
^D
=> ["\nThese lines", "\ncontain a secret", "\nmessage."]
```

We'll come back to the enumerable behaviors of IO objects in the context of file handling in section 12.2. Meanwhile, the three basic IO objects—STDIN, STDOUT, and STDERR—are worth a closer look.

12.1.3 STDIN, STDOUT, STDERR

If you've written programs and/or shell scripts that use any kind of I/O piping, then you're probably familiar with the concept of the *standard* input, output, and error streams. They're basically defaults: unless told otherwise, Ruby assumes that all input will come from the keyboard, and all normal output will go to the terminal. *Assuming*, in this context, means that the unadorned, procedural I/O methods, like puts and gets, operate on STDOUT and STDIN, respectively.

Error messages and STDERR are a little more involved. Nothing goes to STDERR unless someone tells it to. So if you want to use STDERR for output, you have to name it explicitly:

```
if broken?
  STDERR.puts "There's a problem!"
end
```

In addition to the three constants, Ruby also gives you three global variables: $stdin, $stdout, and $stderr.

THE STANDARD I/O GLOBAL VARIABLES

The main difference between STDIN and $stdin (and the other pairs likewise) is that you're not supposed to reassign to the constant but you can reassign to the variable. The variables give you a way to modify default standard I/O stream behaviors without losing the original streams.

For example, perhaps you want all output going to a file, including standard out and standard error. You can achieve this with some assignments to the global variables. Save this code in outputs.rb:

```
record = File.open("/tmp/record", "w")
old_stdout = $stdout
$stdout = record
$stderr = $stdout
puts "This is a record"
z = 10/0
```

The first step is to open the file to which you want to write. (If you don't have a /tmp directory on your system, you can change the filename so that it points to a different path, as long as you have write permission to it.) Next, save the current $stdout to a variable, in case you want to switch back to it later.

Now comes the little dance of the I/O handles. First, $stdout is redefined as the output handle record. Next, $stderr is set equivalent to $stdout. At this point, any plain-old puts statement results in output being written to the file /tmp/record, because plain puts statements output to $stdout—and that's where $stdout is now pointing. $stderr output (like the error message resulting from a division by zero) also goes to the file, because $stderr, too, has been reassigned that file handle.

The result is that when you run the program, you see nothing on your screen; but /tmp/record looks like this:

```
This is a record
outputs.rb:6:in `/': divided by 0 (ZeroDivisionError)
    from outputs.rb:6:in `<main>'
```

Of course, you can also send standard output to one file and standard error to another. The global variables let you manipulate the streams any way you need to.

We'll move on to files soon, but while we're talking about I/O in general and the standard streams in particular, let's look more closely at the keyboard.

12.1.4 A little more about keyboard input

Keyboard input is accomplished, for the most part, with `gets` and `getc`. As you've seen, `gets` returns a single line of input. `getc` returns one character.

One difference between these two methods is that in the case of `getc`, you need to name your input stream explicitly:

```
line = gets
char = STDIN.getc
```

In both cases, input is buffered: you have to press Enter before anything happens. It's possible to make `getc` behave in an unbuffered manner so that it takes its input as soon as the character is struck, but there's no portable way to do this across Ruby platforms. (On UNIX-ish platforms, you can set the terminal to "raw" mode with the `stty` command. You need to use the `system` method, described in chapter 14, to do this from inside Ruby.)

If for some reason you've got `$stdin` set to something other than the keyboard, you can still read keyboard input by using `STDIN` explicitly as the receiver of `gets`:

```
line = STDIN.gets
```

Assuming you've followed the advice in the previous section and done all your standard I/O stream juggling through the use of the global variables rather than the constants, `STDIN` will still be the keyboard input stream, even if `$stdin` isn't.

At this point, we're going to turn to Ruby's facilities for reading, writing, and manipulating files.

12.2 Basic file operations

The built-in class `File` provides the facilities for manipulating files in Ruby. `File` is a subclass of `IO`, so `File` objects share certain properties with `IO` objects, although the `File` class adds and changes certain behaviors.

We'll look first at basic file operations, including opening, reading, writing, and closing files in various modes. Then, we'll look at a more "Rubyish" way to handle file reading and writing: with code blocks. After that, we'll go more deeply into the enumerability of files, and then end the section with an overview of some of the common exceptions and error messages you may get in the course of manipulating files.

12.2.1 The basics of reading from files

Reading from a file can be performed one byte at a time, a specified number of bytes at a time, or one line at a time (where *line* is defined by the `$/` delimiter). You can also change the position of the next read operation in the file by moving forward or backward a certain number of bytes or by advancing the `File` object's internal pointer to a specific byte offset in the file.

All of these operations are performed courtesy of `File` objects. So, the first step is to create a `File` object. The simplest way to do this is with `File.new`. Pass a filename to this constructor, and, assuming the file exists, you'll get back a file handle opened for reading. The following examples involve a file called ticket2.rb that contains the code in listing 3.2 and that's stored in a directory called code:

```
>> f = File.new("code/ticket2.rb")
=> #<File:code/ticket2.rb>
```

(If the file doesn't exist, an exception will be raised.) At this point, you can use the file instance to read from the file. A number of methods are at your disposal. The absolute simplest is the `read` method; it reads in the entire file as a single string:

```
>> f.read
=> "class Ticket\n  def initialize(venue, date)\n
          @venue = venue\n    @date = date\n  end\n\n etc.
```

Although using `read` is tempting in many situations and appropriate in some, it can be inefficient and a bit sledgehammer-like when you need more granularity in your data reading and processing.

We'll look here at a large selection of Ruby's file-reading methods, handling them in groups: first line-based read methods and then byte-based read methods.

> **Close your file handles**
>
> When you're finished reading from and/or writing to a file, you need to close it. `File` objects have a `close` method (for example, `f.close`) for this purpose. You'll learn about a way to open files so that Ruby handles the file closing for you, by scoping the whole file operation to a code block. But if you're doing it the old-fashioned way, as in the examples involving `File.new` in this part of the chapter, you should close your files explicitly. (They'll get closed when you exit irb too, but it's good practice to close the ones you've opened.)

12.2.2 Line-based file reading

The easiest way to read the next line from a file is with `gets`:

```
>> f.gets
=> "class Ticket\n"
>> f.gets
=> "  def initialize(venue, date)\n"
>> f.gets
=> "    @venue = venue\n"
```

The `readline` method does much of what `gets` does: it reads one line from the file. The difference lies in how the two methods behave when you try to read beyond the end of a file: `gets` returns `nil`, and `readline` raises a fatal error. You can see the difference if you do a `read` on a `File` object to get to the end of the file and then try the two methods on the object:

```
>> f.read
=> "  def initialize(venue, date)\n    @venue = venue\n
    @date = date\n  end\n\n
    etc.
>> f.gets
=> nil
>> f.readline
EOFError: end of file reached
```

If you want to get the entire file at once as an array of lines, use `readlines` (a close relative of `read`). Note also the `rewind` operation, which moves the `File` object's internal position pointer back to the beginning of the file:

```
>> f.rewind
=> 0
>> f.readlines
=> ["class Ticket\n", "  def initialize(venue, date)\n",
    "    @venue = venue\n", "    @date = date\n"  etc.
```

Keep in mind that `File` objects are enumerable. That means you can iterate through the lines one at a time rather than reading the whole file into memory. The `each` method of `File` objects (also known by the synonym `each_line`) serves this purpose:

```
>> f.each {|line| puts "Next line: #{line}" }
Next line: class Ticket
Next line:   def initialize(venue, date)
Next line:     @venue = venue
etc.
```

> **NOTE** In the previous example and several that follow, a rewind of the `File` object is assumed. If you're following along in irb, you'll want to type `f.rewind` to get back to the beginning of the file.

The enumerability of `File` objects merits a discussion of its own, and we'll look at it shortly. Meanwhile, let's look at byte-wise simple read operations.

12.2.3 Byte- and character-based file reading

If an entire line is too much, how about one character? The `getc` method reads and returns one character from the file:

```
>> f.getc
=> "c"
```

You can also "un-get" a character—that is, put a specific character back onto the file-input stream so it's the first character read on the next read:

```
>> f.getc
=> "c"
```

```
>> f.ungetc("X")
=> nil
>> f.gets
=> "Xlass Ticket\n"
```

Every character is represented by one or more bytes. How bytes map to characters depends on the encoding. Whatever the encoding, you can move byte-wise as well as character-wise through a file, using `getbyte`. Depending on the encoding, the number of bytes and the number of characters in your file may or may not be equal, and `getc` and `getbyte`, at a given position in the file, may or may not return the same thing.

Just as `readline` differs from `gets` in that `readline` raises a fatal error if you use it at the end of a file, the methods `readchar` and `readbyte` differ from `getc` and `getbyte`, respectively, in the same way. Assuming you've already read to the end of the `File` object `f`, you get the following results:

```
>> f.getc
=> nil
>> f.readchar
EOFError: end of file reached
>> f.getbyte
=> nil
>> f.readbyte
EOFError: end of file reached
```

During all these operations, the `File` object (like any `IO` object) has a sense of where it is in the input stream. As you've seen, you can easily rewind this internal pointer to the beginning of the file. You can also manipulate the pointer in some more fine-grained ways.

12.2.4 Seeking and querying file position

The `File` object has a sense of where in the file it has left off reading. You can both read and change this internal pointer explicitly, using the `File` object's `pos` (position) attribute and/or the `seek` method.

With `pos`, you can tell where in the file the pointer is currently pointing:

```
>> f.rewind
=> 0
>> f.pos
=> 0
>> f.gets
=> "class Ticket\n"
>> f.pos
=> 13
```

Here, the position is `0` after a rewind and `13` after a reading of one 13-byte line. You can assign to the position value, which moves the pointer to a specific location in the file:

```
>> f.pos = 10
=> 10
>> f.gets
=> "et\n"
```

The string returned is what the `File` object considers a "line" as of byte 10: everything from that position onward until the next occurrence of newline (or, strictly speaking, of `$/`).

The `seek` method lets you move around in a file by moving the position pointer to a new location. The location can be a specific offset into the file, or it can be relative to either the current pointer position or the end of the file. You specify what you want using special constants from the `IO` class:

```
f.seek(20, IO::SEEK_SET)
f.seek(15, IO::SEEK_CUR)
f.seek(-10, IO::SEEK_END)
```

In this example, the first line seeks to byte 20. The second line advances the pointer 15 bytes from its current position, and the last line seeks to 10 bytes before the end of the file. Using `IO::SEEK_SET` is optional; a plain `f.seek(20)` does the same thing (as does `f.pos = 20`).

We've looked at several ways to read from files, starting with the all-at-once read method, progressing through the line-by-line approach, and winding up with the most fine-grained reads based on character and position. All of these file-reading techniques involve `File` objects—that is, instances of the `File` class. That class itself also offers some reading techniques.

12.2.5 Reading files with File class methods

A little later, you'll see more of the facilities available as class methods of `File`. For now, we'll look at two methods that handle file reading at the class level: `File.read` and `File.readlines`.

These two methods do the same thing their same-named instance-method counterparts do; but instead of creating an instance, you use the `File` class, the method name, and the name of the file:

```
full_text = File.read("myfile.txt")
lines_of_text = File.readlines("myfile.txt")
```

In the first case, you get a string containing the entire contents of the file. In the second case, you get an array of lines.

These two class methods exist purely for convenience. They take care of opening and closing the file handle for you; you don't have to do any system-level housekeeping. Most of the time, you'll want to do something more complex and/or more efficient than reading the entire contents of a file into a string or an array at one time. Given that even the `read` and `readlines` instance methods are relatively coarse-grained tools, if you decide to read a file in all at once, you may as well go all the way and use the class-method versions.

You now have a good toolkit for reading files and dealing with the results. At this point, we'll turn to the other side of the equation: writing to files.

Low-level I/O methods

In addition to the various I/O and `File` methods we'll look at closely here, the `IO` class gives you a toolkit of system-level methods with which you can do low-level I/O operations. These include `sysseek`, `sysread`, and `syswrite`. These methods correspond to the system calls on which some of the higher-level methods are built.

The `sys-` methods perform raw, unbuffered data operations and shouldn't be mixed with higher-level methods. Here's an example of what not to do:

```
File.open("output.txt", "w") do |f|
  f.print("Hello")
  f.syswrite(" there!")
end
puts File.read("output.txt")
```

If you run this little program, here's what you'll see:

```
syswrite.rb:3: warning: syswrite for buffered IO
 there!Hello
```

In addition to a warning, you get the second string (the one written with `syswrite`) stuck in the file before the first string. That's because `syswrite` and `print` don't operate according to the same rules and don't play nicely together. It's best to stick with the higher-level methods unless you have a particular reason to use the others.

12.2.6 Writing to files

Writing to a file involves using `puts`, `print`, or `write` on a `File` object that's opened in write or append mode. Write mode is indicated by `w` as the second argument to `new`. In this mode, the file is created (assuming you have permission to create it); if it existed already, the old version is overwritten. In append mode (indicated by `a`), whatever you write to the file is appended to what's already there. If the file doesn't exist yet, opening it in append mode creates it.

This example performs some simple write and append operations, pausing along the way to use the mighty `File.read` to check the contents of the file:

```
>> f = File.new("data.out", "w")
=> #<File:data.out>
>> f.puts "David A. Black, Rubyist"
=> nil
>> f.close
=> nil
>> puts File.read("data.out")
David A. Black, Rubyist
=> nil
>> f = File.new("data.out", "a")
=> #<File:data.out>
>> f.puts "Yukihiro Matsumoto, Ruby creator"
=> nil
>> f.close
=> nil
```

```
>> puts File.read("data.out")
David A. Black, Rubyist
Yukihiro Matsumoto, Ruby creator
```

The return value of a call to `puts` on a `File` object is the same as the return value of any call to `puts`: `nil`. The same is true of `print`. If you use the lower-level `write` method, which is an instance method of the `IO` class (and therefore available to `File` objects, because `File` inherits from `IO`), the return value is the number of bytes written to the file.

Ruby lets you economize on explicit closing of `File` objects—and enables you to keep your code nicely encapsulated—by providing a way to perform file operations inside a code block. We'll look at this elegant and common technique next.

12.2.7 Using blocks to scope file operations

Using `File.new` to create a `File` object has the disadvantage that you end up having to close the file yourself. Ruby provides an alternate way to open files that puts the housekeeping task of closing the file in the hands of Ruby: `File.open` with a code block.

If you call `File.open` with a code block, the block receives the `File` object as its single argument. You use that `File` object inside the block. When the block ends, the `File` object is automatically closed.

Here's an example in which a file is opened and read in line by line for processing. First, create a file called records.txt containing one record per line:

```
Pablo Casals|Catalan|cello|1876-1973
Jascha Heifetz|Russian-American|violin|1901-1988
Emanuel Feuermann|Austrian-American|cello|1902-1942
```

Now write the code that will read this file, line by line, and report on what it finds. It uses the block-based version of `File.open`:

```
File.open("records.txt") do |f|
  while record = f.gets
    name, nationality, instrument, dates = record.chomp.split('|')
    puts "#{name} (#{dates}), who was #{nationality},
➥ played #{instrument}. "
  end
end
```

The program consists entirely of a call to `File.open` along with its code block. (If you call `File.open` without a block, it acts like `File.new`.) The block parameter, `f`, receives the `File` object. Inside the block, the file is read one line at a time using `f`. The `while` test succeeds as long as lines are coming in from the file. When the program hits the end of the input file, `gets` returns `nil`, and the `while` condition fails.

Inside the `while` loop, the current line is chomped so as to remove the final newline character, if any, and split on the pipe character. The resulting values are stored in the four local variables on the left, and those variables are then interpolated into a pretty-looking report for output:

```
Pablo Casals (1876-1973), who was Catalan, played cello.
Jascha Heifetz (1901-1988), who was Russian-American, played violin.
Emanuel Feuermann (1902-1942), who was Austrian-American, played cello.
```

The use of a code block to scope a `File.open` operation is common. It sometimes leads to misunderstandings, though. In particular, remember that the block that provides you with the `File` object doesn't do anything else. There's no implicit loop. If you want to read what's in the file, you still have to do something like a `while` loop using the `File` object. It's just nice that you get to do it inside a code block and that you don't have to worry about closing the `File` object afterward.

And don't forget that `File` objects are enumerable.

12.2.8 File enumerability

Thanks to the fact that `Enumerable` is among the ancestors of `File`, you can replace the `while` idiom in the previous example with `each`:

```
File.open("records.txt") do |f|
  f.each do |record|
    name, nationality, instrument, dates = record.chomp.split('|')
    puts "#{name} (#{dates}), who was #{nationality},
 ➥ played #{instrument}. "
  end
end
```

Ruby gracefully stops iterating when it hits the end of the file.

As enumerables, `File` objects can perform many of the same functions that arrays, hashes, and other collections do. Understanding how file enumeration works requires a slightly different mental model: whereas an array exists already and walks through its elements in the course of iteration, `File` objects have to manage line-by-line reading behind the scenes when you iterate through them. But the similarity of the idioms—the common use of the methods from `Enumerable`—means you don't have to think in much detail about the file-reading process when you iterate through a file.

Most important, don't forget that you can iterate through files and address them as enumerables. It's tempting to read a whole file into an array and then process the array. But why not just iterate on the file and avoid wasting the space required to hold the file's contents in memory?

You could, for example, read in an entire file of plain-text records and then perform an inject operation on the resulting array to get the average of a particular field:

```
# Sample record in members.txt:
# David Black male 55
count = 0
total_ages = File.readlines("members.txt").inject(0) do |total,line|
  count += 1
  fields = line.split
  age = fields[3].to_i
  total + age
end
puts "Average age of group: #{total_ages / count}."
```

But you can also perform the inject operation directly on the File object:

```
count = 0
total_ages = File.open("members.txt") do |f|
  f.inject(0) do |total,line|

    count += 1
    fields = line.split
    age = fields[3].to_i
    total + age
  end
end
puts "Average age of group: #{total_ages / count}."
```

With this approach, no intermediate array is created. The File object does its own work.

One way or another, you'll definitely run into cases where something goes wrong with your file operations. Ruby will leave you in no doubt that there's a problem, but it's helpful to see in advance what some of the possible problems are and how they're reported.

12.2.9 File I/O exceptions and errors

When something goes wrong with file operations, Ruby raises an exception. Most of the errors you'll get in the course of working with files can be found in the Errno namespace: Errno::EACCES (permission denied), Errno::ENOENT (no such entity—a file or directory), Errno:EISDIR (is a directory—an error you get when you try to open a directory as if it were a file), and others. You'll always get a message along with the exception:

```
>> File.open("no_file_with_this_name")
Errno::ENOENT: No such file or directory - no_file_with_this_name
>> f = File.open("/tmp")
=> #<File:/tmp>
>> f.gets
Errno::EISDIR: Is a directory - /tmp
>> File.open("/var/root")
Errno::EACCES: Permission denied - /var/root
```

The Errno family of errors includes not only file-related errors but also other system errors. The underlying system typically maps errors to integers (for example, on Linux, the "not a directory" error is represented by the C macro ENOTDIR, which is defined as the number 20). Ruby's Errno class wraps these error-to-number mappings in a bundle of exception classes.

Each Errno exception class contains knowledge of the integer to which its corresponding system error maps. You can get these numbers via the Errno constant of each Errno class—and if that sounds obscure, an example will make it clearer:

```
>> Errno::ENOTDIR::Errno
=> 20
```

You'll rarely, if ever, have to concern yourself with the mapping of Ruby's Errno exception classes to the integers to which your operating system maps errors. But you should be aware that any Errno exception is basically a system error percolating up through Ruby.

These aren't Ruby-specific errors, like syntax errors or missing method errors; they involve things going wrong at the system level. In these situations, Ruby is just the messenger.

Let's go back to what you can do when things go right. We'll look next at some ways in which you can ask `IO` and `File` objects for information about themselves and their state.

12.3 Querying IO and File objects

`IO` and `File` objects can be queried on numerous criteria. The `IO` class includes some query methods; the `File` class adds more.

One class and one module closely related to `File` also get into the act: `File::Stat` and `FileTest`. `File::Stat` returns objects whose attributes correspond to the fields of the stat structure defined by the C library call `stat(2)`. Some of these fields are system-specific and not meaningful on all platforms. The `FileTest` module offers numerous methods for getting status information about files.

The `File` class also has some query methods. In some cases, you can get the same information about a file several ways:

```
>> File.size("code/ticket2.rb")
=> 219
>> FileTest.size("code/ticket2.rb")
=> 219
>> File::Stat.new("code/ticket2.rb").size
=> 219
```

In what follows, we'll look at a large selection of query methods. In some cases, they're available in more than one way.

12.3.1 Getting information from the File class and the FileTest module

`File` and `FileTest` offer numerous query methods that can give you lots of information about a file. These are the main categories of query: *What is it? What can it do? How big is it?*

The methods available as class methods of `File` and `FileTest` are almost identical; they're mostly aliases of each other. The examples will only use `FileTest`, but you can use `File` too.

Here are some questions you might want to ask about a given file, along with the techniques for asking them. All of these methods return either true or false except `size`, which returns an integer. Keep in mind that these file-testing methods are happy to take directories, links, sockets, and other filelike entities as their arguments. They're not restricted to regular files:

- *Does a file exist?*

  ```
  FileTest.exist?("/usr/local/src/ruby/README")
  ```

- *Is the file a directory? A regular file? A symbolic link?*

  ```
  FileTest.directory?("/home/users/dblack/info")
  FileTest.file?("/home/users/dblack/info")
  FileTest.symlink?("/home/users/dblack/info")
  ```

This family of query methods also includes `blockdev?`, `pipe?`, `chardev?`, and `socket?`.

- *Is a file readable? Writable? Executable?*

```
FileTest.readable?("/tmp")
FileTest.writable?("/tmp")
FileTest.executable?("/home/users/dblack/setup")
```

This family of query methods includes `world_readable?` and `world_writable?`, which test for more permissive permissions. It also includes variants of the basic three methods with `_real` appended. These test the permissions of the script's actual runtime ID as opposed to its effective user ID.

- *What is the size of this file? Is the file empty (zero bytes)?*

```
FileTest.size("/home/users/dblack/setup")
FileTest.zero?("/tmp/tempfile")
```

Getting file information with Kernel#test

Among the top-level methods at your disposal (that is, private methods of the `Kernel` module, which you can call anywhere without a receiver, like `puts`) is a method called `test`. You use `test` by passing it two arguments: the first represents the test, and the second is a file or directory. The choice of test is indicated by a character. You can represent the value using the `?c` notation, where `c` is the character, or as a one-character string.

Here's an example that finds out whether `/tmp` exists:

```
test ?e, "/tmp"
```

Other common test characters include `?d` (the test is true if the second argument is a directory), `?f` (true if the second argument is a regular file), and `?z` (true if the second argument is a zero-length file). For every test available through `Kernel#test`, there's usually a way to get the result by calling a method of one of the classes discussed in this section. But the `Kernel#test` notation is shorter and can be handy for that reason.

In addition to the query and Boolean methods available through `FileTest` (and `File`), you can also consult objects of the `File::Stat` class for file information.

12.3.2 Deriving file information with File::Stat

`File::Stat` objects have attributes corresponding to the stat structure in the standard C library. You can create a `File::Stat` object in either of two ways: with the `new` method or with the `stat` method on an existing `File` object:

```
>> File::Stat.new("code/ticket2.rb")
=> #<File::Stat dev=0x1000002, ino=11531534, mode=0100644,
nlink=1, uid=501, gid=20, rdev=0x0, size=219, blksize=4096,
blocks=8, atime=2014-03-23 08:31:49 -0400,
mtime=2014-02-25 06:24:43 -0500, ctime=2014-02-25 06:24:43 -0500>

>> File.open("code/ticket2.rb") {|f| f.stat }
```

Same output

The screen output from the `File::Stat.new` method shows you the attributes of the object, including its times of creation (`ctime`), last modification (`mtime`), and last access (`atime`).

> **TIP** The code block given to `File.open` in this example, `{|f| f.stat }`, evaluates to the last expression inside it. Because the last (indeed, only) expression is `f.stat`, the value of the block is a `File::Stat` object. In general, when you use `File.open` with a code block, the call to `File.open` returns the last value from the block. Called without a block, `File.open` (like `File.new`) returns the newly created `File` object.

Much of the information available from `File::Stat` is built off of UNIX-like metrics, such as inode number, access mode (permissions), and user and group ID. The relevance of this information depends on your operating system. We won't go into the details here because it's not cross-platform; but whatever information your system maintains about files is available if you need it.

Manipulating and querying files often involves doing likewise to directories. Ruby provides facilities for directory operations in the `Dir` class. You'll also see such operations in some of the standard library tools we'll discuss a little later. First, let's look at `Dir`.

12.4 Directory manipulation with the Dir class

Like `File`, the `Dir` class provides useful class and instance methods. To create a `Dir` instance, you pass a directory path to `new`:

```
>> d = Dir.new("/usr/local/src/ruby/lib/minitest")
=> #<Dir:/usr/local/src/ruby/lib/minitest>
```

Adjust path as needed for your system

The most common and useful `Dir`-related technique is iteration through the entries (files, links, other directories) in a directory.

12.4.1 Reading a directory's entries

You can get hold of the entries in one of two ways: using the `entries` method or using the `glob` technique. The main difference is that *globbing* the directory doesn't return hidden entries, which on many operating systems (including all UNIX-like systems) means entries whose names start with a period. Globbing also allows for wildcard matching and for recursive matching in subdirectories.

THE ENTRIES METHOD

Both the `Dir` class itself and instances of the `Dir` class can give you a directory's entries. Given the instance of `Dir` created earlier, you can do this:

```
>> d.entries
=> [".", "..", ".document", "autorun.rb", "benchmark.rb", "hell.rb",
"mock.rb", "parallel_each.rb", "pride.rb", "README.txt", "spec.rb",
"unit.rb"]
```

Or you can use the class-method approach:

```
>> Dir.entries("/usr/local/src/ruby/lib/minitest")
=> [".", "..", ".document", "autorun.rb", "benchmark.rb", "hell.rb",
"mock.rb", "parallel_each.rb", "pride.rb", "README.txt", "spec.rb",
"unit.rb"]
```

Note that the single- and double-dot entries (current directory and parent directory, respectively) are present, as is the hidden `.document` entry. If you want to iterate through the entries, only processing files, you need to make sure you filter out the names starting with dots.

Let's say we want to add up the sizes of all non-hidden regular files in a directory. Here's a first iteration (we'll develop a shorter one later):

```
d = Dir.new("/usr/local/src/ruby/lib/minitest")
entries = d.entries
entries.delete_if {|entry| entry =~ /^\./ }
entries.map! {|entry| File.join(d.path, entry) }
entries.delete_if {|entry| !File.file?(entry) }
print "Total bytes: "
puts entries.inject(0) {|total, entry| total + File.size(entry) }
```

First, we create a `Dir` object for the target directory and grab its entries. Next comes a sequence of manipulations on the array of entries. Using the `delete_if` array method, we remove all that begin with a dot. Then, we do an in-place mapping of the entry array so that each entry includes the full path to the file. This is accomplished with two useful methods: the instance method `Dir#path`, which returns the original directory path underlying this particular `Dir` instance (/usr/local/src/ruby/lib/minitest); and `File.join`, which joins the path to the filename with the correct separator (usually /, but it's somewhat system-dependent).

Now that the entries have been massaged to represent full pathnames, we do another `delete_if` operation to delete all the entries that aren't regular files, as measured by the `File.file?` test method. The entries array now contains full pathnames of all the regular files in the original directory. The last step is to add up their sizes, a task for which `inject` is perfectly suited.

Among other ways to shorten this code, you can use directory globbing instead of the `entries` method.

DIRECTORY GLOBBING

Globbing in Ruby takes its semantics largely from shell globbing, the syntax that lets you do things like this in the shell:

```
$ ls *.rb
$ rm *.?xt
$ for f in [A-Z]*   # etc.
```

The details differ from one shell to another, of course; but the point is that this whole family of name-expansion techniques is where Ruby gets its globbing syntax. An asterisk represents a wildcard match on any number of characters; a question mark represents one wildcard character. Regexp-style character classes are available for matching.

To glob a directory, you can use the `Dir.glob` method or `Dir.[]` (square brackets). The square-bracket version of the method allows you to use index-style syntax, as you would with the square-bracket method on an array or hash. You get back an array containing the result set:

```
>> Dir["/usr/local/src/ruby/include/ruby/r*.h"]
=> ["/usr/local/src/ruby/include/ruby/re.h", "/usr/local/src/ruby/include/
     ruby/regex.h", "/usr/local/src/ruby/include/ruby/ruby.h"]
```

The `glob` method is largely equivalent to the `[]` method but a little more versatile: you can give it not only a glob pattern but also one or more flag arguments that control its behavior. For example, if you want to do a case-insensitive glob, you can pass the `File::FNM_CASEFOLD` flag:

```
Dir.glob("info*")      # []
Dir.glob("info", File::FNM_CASEFOLD   # ["Info", "INFORMATION"]
```

Another useful flag is `FNM_DOTMATCH`, which includes hidden dot files in the results.

If you want to use two flags, you combine them with the bitwise OR operator, which consists of a single pipe character. In this example, progressively more files are found as the more permissive flags are added:

```
>> Dir.glob("*info*")
=> []
>> Dir.glob("*info*", File::FNM_DOTMATCH)
=> [".information"]
>> Dir.glob("*info*", File::FNM_DOTMATCH | File::FNM_CASEFOLD)
=> [".information", ".INFO", "Info"]
```

The flags are, literally, numbers. The value of `File::FNM_DOTMATCH`, for example, is `4`. The specific numbers don't matter (they derive ultimately from the flags in the system library function `fnmatch`). What does matter is the fact that they're exponents of two accounts for the use of the OR operation to combine them.

> **NOTE** As you can see from the first two lines of the previous example, a `glob` operation on a directory can find nothing and still not complain. It gives you an empty array. Not finding anything isn't considered a failure when you're globbing.

Globbing with square brackets is the same as globbing without providing any flags. In other words, doing this

```
Dir["*info*"]
```

is like doing this

```
Dir.glob("*info*", 0)
```

which, because the default is that none of the flags is in effect, is like doing this:

```
Dir.glob("*info*")
```

The square-bracket method of `Dir` gives you a kind of shorthand for the most common case. If you need more granularity, use `Dir.glob`.

By default, globbing doesn't include filenames that start with dots. Also, as you can see, globbing returns full pathnames, not just filenames. Together, these facts let us trim down the file-size totaling example:

```
dir = "/usr/local/src/ruby/lib/minitest"
entries = Dir["#{dir}/*"].select {|entry| File.file?(entry) }
print "Total bytes: "
puts entries.inject(0) {|total, entry| total + File.size(entry) }
```

With their exclusion of dot files and their inclusion of full paths, glob results often correspond more closely than `Dir.entries` results to the ways that many of us deal with files and directories on a day-to-day basis.

There's more to directory management than just seeing what's there. We'll look next at some techniques that let you go more deeply into the process.

12.4.2 Directory manipulation and querying

The `Dir` class includes several query methods for getting information about a directory or about the current directory, as well as methods for creating and removing directories. These methods are, like so many, best illustrated by example.

Here, we'll create a new directory (`mkdir`), navigate to it (`chdir`), add and examine a file, and delete the directory (`rmdir`):

```
newdir = "/tmp/newdir"                          ❶
newfile = "newfile"
Dir.mkdir(newdir)
Dir.chdir(newdir) do                            ❷
  File.open(newfile, "w") do |f|
    f.puts "Sample file in new directory"       ❸
  end
  puts "Current directory: #{Dir.pwd}"          ❹
  puts "Directory listing: "
  p Dir.entries(".")
  File.unlink(newfile)                          ❺
end
Dir.rmdir(newdir)                               ❻
print "Does #{newdir} still exist? "
if File.exist?(newdir)                          ❼
  puts "Yes"
else
  puts "No"
end
```

After initializing a couple of convenience variables ❶, we create the new directory with `mkdir`. With `Dir.chdir`, we change to that directory; also, using a block with `chdir` means that after the block exits, we're back in the previous directory ❷. (Using `chdir` without a block changes the current directory until it's explicitly changed back.)

As a kind of token directory-populating step, we create a single file with a single line in it ❸. We then examine the current directory name using `Dir.pwd` and look at a

listing of the entries in the directory ❹. Next, we unlink (delete) the recently created file ❺, at which point the `chdir` block is finished.

Back in whatever directory we started in, we remove the sample directory using `Dir.rmdir` (also callable as `unlink` or `delete`) ❻. Finally, we test for the existence of `newdir`, fully expecting an answer of `No` (because `rmdir` would have raised a fatal error if it hadn't found the directory and successfully removed it) ❼.

As promised in the introduction to this chapter, we'll now look at some standard library facilities for manipulating and handling files.

12.5 File tools from the standard library

File handling is an area where the standard library's offerings are particularly rich. Accordingly, we'll delve into those offerings more deeply here than anywhere else in the book. This isn't to say that the rest of the standard library isn't worth getting to know, but that the extensions available for file manipulation are so central to how most people do file manipulation in Ruby that you can't get a firm grounding in the process without them.

We'll look at the versatile `FileUtils` package first and then at the more specialized but useful `Pathname` class. Next you'll meet `StringIO`, a class whose objects are, essentially, strings with an I/O interface; you can `rewind` them, `seek` through them, `getc` from them, and so forth. Finally, we'll explore `open-uri`, a package that lets you "open" URIs and read them into strings as easily as if they were local files.

12.5.1 The FileUtils module

The `FileUtils` module provides some practical and convenient methods that make it easy to manipulate files from Ruby in a concise manner in ways that correspond to familiar system commands. The methods' names will be particularly familiar to users of UNIX and UNIX-like operating systems. They can be easily learned by those who don't know them already.

Many of the methods in `FileUtils` are named in honor of system commands with particular command-line options. For example, `FileUtils.rm_rf` emulates the `rm -rf` command (force unconditional recursive removal of a file or directory). You can create a symbolic link from *filename* to *linkname* with `FileUtils.ln_s(filename, linkname)`, much in the manner of the `ln -s` command.

As you can see, some of the methods in `FileUtils` are operating-system specific. If your system doesn't support symbolic links, then `ln_s` won't work. But the majority of the module's methods are portable. We'll look here at examples of some of the most useful ones.

COPYING, MOVING, AND DELETING FILES

`FileUtils` provides several concise, high-level methods for these operations. The `cp` method emulates the traditional UNIX method of the same name. You can `cp` one file to another or several files to a directory:

```
>> require 'fileutils'
=> true
>> FileUtils.cp("baker.rb", "baker.rb.bak")
=> nil
>> FileUtils.mkdir("backup")                          <-- 1
=> ["backup"]
>> FileUtils.cp(["ensure.rb", "super.rb"], "backup")
=> ["ensure.rb", "super.rb"]
>> Dir["backup/*"]                                    <-- 2
=> ["backup/ensure.rb", "backup/super.rb"]
```

This example also illustrates the `mkdir` method ❶ as well as the use of `Dir#[]` ❷ to verify the presence of the copied files in the new backup directory.

Just as you can copy files, you can also move them, individually or severally:

```
>> FileUtils.mv("baker.rb.bak", "backup")
=> 0
>> Dir["backup/*"]
=> ["backup/baker.rb.bak", "backup/ensure.rb", "backup/super.rb"]
```

And you can remove files and directories easily:

```
>> File.exist?("backup/super.rb")
=> true
>> FileUtils.rm("./backup/super.rb")
=> ["./backup/super.rb"]
>> File.exist?("backup/super.rb")
=> false
```

The `rm_rf` method recursively and unconditionally removes a directory:

```
>> FileUtils.rm_rf("backup")
=> ["backup"]
>> File.exist?("backup")
=> false
```

`FileUtils` gives you a useful toolkit for quick and easy file maintenance. But it goes further: it lets you try commands without executing them.

THE DRYRUN AND NOWRITE MODULES

If you want to see what would happen if you were to run a particular `FileUtils` command, you can send the command to `FileUtils::DryRun`. The output of the method you call is a representation of a UNIX-style system command, equivalent to what you'd get if you called the same method on `FileUtils`:

```
>> FileUtils::DryRun.rm_rf("backup")
rm -rf backup
=> nil
>> FileUtils::DryRun.ln_s("backup", "backup_link")
ln -s backup backup_link
=> nil
```

If you want to make sure you don't accidentally delete, overwrite, or move files, you can give your commands to `FileUtils::NoWrite`, which has the same interface as `FileUtils` but doesn't perform any disk-writing operations:

```
>> FileUtils::NoWrite.rm("backup/super.rb")
=> nil
>> File.exist?("backup/super.rb")
=> true
```

You'll almost certainly find `FileUtils` useful in many situations. Even if you're not familiar with the UNIX-style commands on which many of `FileUtils`'s method names are based, you'll learn them quickly, and it will save you having to dig deeper into the lower-level I/O and file libraries to get your tasks done.

Next we'll look at another file-related offering from the standard library: the pathname extension.

12.5.2 The Pathname class

The `Pathname` class lets you create `Pathname` objects and query and manipulate them so you can determine, for example, the basename and extension of a pathname, or iterate through the path as it ascends the directory structure.

`Pathname` objects also have a large number of methods that are proxied from `File`, `Dir`, `IO`, and other classes. We won't look at those methods here; we'll stick to the ones that are uniquely `Pathname`'s.

First, start with a `Pathname` object:

```
>> require 'pathname'
=> true
>> path = Pathname.new("/Users/dblack/hacking/test1.rb")
=> #<Pathname:/Users/dblack/hacking/test1.rb>
```

When you call methods on a `Pathname` object, you often get back another `Pathname` object. But the new object always has its string representation visible in its own `inspect` string. If you want to see the string on its own, you can use `to_s` or do a `puts` on the pathname.

Here are two ways to examine the basename of the path:

```
>> path.basename
=> #<Pathname:test1.rb>
>> puts path.basename
test1.rb
```

You can also examine the directory that contains the file or directory represented by the pathname:

```
>> path.dirname
=> #<Pathname:/Users/dblack/hacking>
```

If the last segment of the path has an extension, you can get the extension from the `Pathname` object:

```
>> path.extname
=> ".rb"
```

The `Pathname` object can also walk up its file and directory structure, truncating itself from the right on each iteration, using the `ascend` method and a code block:

```
>> path.ascend do |dir|
?>   puts "Next level up: #{dir}"
>> end
```

Here's the output:

```
Next level up: /Users/dblack/hacking/test1.rb
Next level up: /Users/dblack/hacking
Next level up: /Users/dblack
Next level up: /Users
Next level up: /
```

The key behavioral trait of `Pathname` objects is that they return other `Pathname` objects. That means you can extend the logic of your pathname operations without having to convert back and forth from pure strings. By way of illustration, here's the last example again, but altered to take advantage of the fact that what's coming through in the block parameter `dir` on each iteration isn't a string (even though it prints out like one) but a `Pathname` object:

```
>> path = Pathname.new("/Users/dblack/hacking/test1.rb")
=> #<Pathname:/Users/dblack/hacking/test1.rb>
>> path.ascend do |dir|
?>   puts "Ascended to #{dir.basename}"
>> end
```

The output is

```
Ascended to test1.rb
Ascended to hacking
Ascended to dblack
Ascended to Users
Ascended to /
```

The fact that `dir` is always a `Pathname` object means that it's possible to call the `basename` method on it. It's true that you can always call `File.basename(string)` on any string. But the `Pathname` class pinpoints the particular knowledge that a path might be assumed to encapsulate about itself and makes it available to you via simple method calls.

We'll look next at a different and powerful standard library class: `StringIO`.

12.5.3 The StringIO class

The `StringIO` class allows you to treat strings like `IO` objects. You can seek through them, rewind them, and so forth.

The advantage conferred by `StringIO` is that you can write methods that use an `IO` object API, and those methods will be able to handle strings. That can be useful for testing, as well as in a number of real runtime situations.

Let's say, for example, that you have a module that decomments a file: it reads from one file and writes everything that isn't a comment to another file. Here's what such a module might look like:

```
module DeCommenter
  def self.decomment(infile, outfile, comment_re = /\A\s*#/)
    infile.each do |inline|
      outfile.print inline unless inline =~ comment_re
    end
  end
end
```

The `DeCommenter.decomment` method expects two open file handles: one it can read from and one it can write to. It also takes a regular expression, which has a default value. The regular expression determines whether each line in the input is a comment. Every line that does *not* match the regular expression is printed to the output file.

A typical use case for the `DeCommenter` module would look like this:

```
File.open("myprogram.rb") do |inf|
  File.open("myprogram.rb.out", "w") do |outf|
    DeCommenter.decomment(inf, outf)
  end
end
```

In this example, we're taking the comments out of the hypothetical program file myprogram.rb.

What if you want to write a test for the `DeCommenter` module? Testing file transformations can be difficult because you need to maintain the input file as part of the test and also make sure you can write to the output file—which you then have to read back in. `StringIO` makes it easier by allowing all of the code to stay in one place without the need to read or write actual files.

Testing using real files

If you want to run tests on file input and output using real files, Ruby's `tempfile` class can help you. It's a standard-library feature, so you have to `require 'tempfile'`. Then, you create temporary files with the constructor, passing in a name that Ruby munges into a unique filename. For example:

```
tf = Tempfile.new("my_temp_file").
```

You can then write to and read from the file using the `File` object `tf`.

To use the decommenter with `StringIO`, save the module to decommenter.rb. Then, create a second file, decomment-demo.rb, in the same directory and with the following contents:

```
require 'stringio'                          ❶
require_relative 'decommenter'
string = <<EOM
# This is a comment.                        ❷
This isn't a comment.
# This is.
   # So is this.
```

```
This is also not a comment.                          ❷
EOM
infile = StringIO.new(string)                        ❸
outfile = StringIO.new("")
DeCommenter.decomment(infile,outfile)                ❹
puts "Test succeeded" if outfile.string == <<EOM     ❺
This isn't a comment.
This is also not a comment.
EOM
```

After loading both the `stringio` library and the decommenter code ❶, the program sets `string` to a five-line string (created using a here-document) containing a mix of comment lines and non-comment lines ❷. Next, two `StringIO` objects are created: one that uses the contents of `string` as its contents, and one that's empty ❸. The empty one represents the output file.

Next comes the call to `DeCommenter.decomment` ❹. The module treats its two arguments as `File` or `IO` objects, reading from one and printing to the other. `StringIO` objects happily behave like `IO` objects, and the filtering takes place between them. When the filtering is done, you can check explicitly to make sure that what was written to the output "file" is what you expected ❺. The original and changed contents are both physically present in the same file, which makes it easier to see what the test is doing and also easier to change it.

Another useful standard library feature is the `open-uri` library.

12.5.4 The open-uri library

The `open-uri` standard library package lets you retrieve information from the network using the HTTP and HTTPS protocols as easily as if you were reading local files. All you do is require the library (`require 'open-uri'`) and use the `Kernel#open` method with a URI as the argument. You get back a `StringIO` object containing the results of your request:

```
require 'open-uri'
rubypage = open("http://rubycentral.org")
puts rubypage.gets
```

You get the `doctype` declaration from the Ruby Central homepage—not the most scintillating reading, but it demonstrates the ease with which `open-uri` lets you import networked materials.

12.6 Summary

In this chapter you've seen

- I/O (keyboa0rd and screen) and file operations in Ruby
- File objects as enumerables
- The `STDIN`, `STDOUT`, and `STDERR` objects
- The `FileUtils` module
- The `Pathname` module

- The `StringIO` class
- The `open-uri` module

I/O operations are based on the `IO` class, of which `File` is a subclass. Much of what `IO` and `File` objects do consists of wrapped library calls; they're basically API libraries that sit on top of system I/O facilities.

You can iterate through Ruby file handles as if they were arrays, using `each`, `map`, `reject`, and other methods from the `Enumerable` module, and Ruby will take care of the details of the file handling. If and when you need to, you can also address `IO` and `File` objects with lower-level commands.

Some of the standard-library facilities for file manipulation are indispensable, and we looked at several: the `FileUtils` module, which provides an enriched toolkit for file and disk operations; the `StringIO` class, which lets you address a string as if it were an I/O stream; the `Pathname` extension, which allows for easy, extended operations on strings representing file-system paths; and `open-uri`, which makes it easy to "open" documents on the network.

We also looked at keyboard input and screen output, which are handled via `IO` objects—in particular, the standard input, output, and error I/O handles. Ruby lets you reassign these so you can redirect input and output as needed.

With this chapter, we've come to the end of part 2 of the book and thus the end of our survey of Ruby built-in features and classes. We'll turn in part 3 to the broad and deep matter of Ruby dynamics, starting with a look at one of the simplest yet most profound premises of Ruby: the premise that objects, even objects of the same class, can act and react individually.

RUBY

THE WELL-GROUNDED Rubyist Second Edition

David A. Black

This is a good time for Ruby! It's powerful like Java or C++, and has dynamic features that let your code react gracefully to changes at runtime. And it's elegant, so creating applications, development tools, and administrative scripts is easier and more straightforward. With the long-awaited Ruby 2, an active development community, and countless libraries and productivity tools, Ruby has come into its own.

The Well-Grounded Rubyist, Second Edition is a beautifully written tutorial that begins with your first Ruby program and goes on to explore sophisticated topics like callable objects, reflection, and threading. The book concentrates on the language, preparing you to use Ruby in any way you choose. This second edition includes coverage of new Ruby features such as keyword arguments, lazy enumerators, and Module#prepend, along with updated information on new and changed core classes and methods.

What's Inside

- Clear explanations of Ruby concepts
- Numerous simple examples
- Updated for Ruby 2.1
- Prepares you to use Ruby anywhere for any purpose

David A. Black is an internationally known Ruby developer, author, trainer, speaker, event organizer, and founder of Ruby Central as well as a Lead Consultant at Cyrus Innovation.

"Once again, the definitive book on Ruby from David Black. A must-have!"
—William Wheeler, TekSystems

"All wheat, no chaff—takes you from Ruby programmer to full-fledged Rubyist."
—Doug Sparling
Andrews McMeel Universal

"Provides powerful insights and digs into Ruby's quirks. Revelatory."
—Ted Roche
Ted Roche & Associates, LLC

"The best way to learn Ruby fundamentals."
—Derek Sivers, sivers.org

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
manning.com/TheWellGroundedRubyistSecondEdition

ISBN 13: 978-1-617291-69-2
ISBN 10: 1-617291-69-2
54499
9 781617 291692

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]