Chapter 11

Basic Input and Output

Ruby provides what at first sight looks like two separate sets of I/O routines. The first is the simple interface—we've been using it pretty much exclusively so far:

```
print "Enter your name: "
name = gets
```

A whole set of I/O-related methods is implemented in the Kernel module—gets, open, print, printf, putc, puts, readline, readlines, and test—that makes it simple and convenient to write straightforward Ruby programs. These methods typically do I/O to standard input and standard output, which makes them useful for writing filters. You'll find them documented starting on page 564.

The second way, which gives you a lot more control, is to use IO objects.

What Is an IO Object?

Ruby defines a single base class, IO, to handle input and output. This base class is subclassed by classes File and BasicSocket to provide more specialized behavior, but the principles are the same. An IO object is a bidirectional channel between a Ruby program and some external resource.[1] An IO object may have more to it than meets the eye, but in the end you still simply write to it and read from it.

In this chapter, we'll be concentrating on class IO and its most commonly used subclass, class File. For more details on using the socket classes for networking, see the section beginning on page 878.

1. For those who just have to know the implementation details, this means that a single IO object can sometimes be managing more than one operating system file descriptor. For example, if you open a pair of pipes, a single IO object contains both a read pipe and a write pipe.

Opening and Closing Files

As you may expect, you can create a new file object using File.new:

```
file = File.new("testfile", "r")
# ... process the file
file.close
```

The first parameter is the filename. The second is the mode string, which lets you open the file for reading, writing, or both. (Here we opened testfile for reading with an "r". We could also have used "w" for write or "r+" for read-write. The full list of allowed modes appears on page 547.) You can also optionally specify file permissions when creating a file; see the description of File.new on page 512 for details. After opening the file, we can work with it, writing and/or reading data as needed. Finally, as responsible software citizens, we close the file, ensuring that all buffered data is written and that all related resources are freed.

But here Ruby can make life a little bit easier for you. The method File.open also opens a file. In regular use, it behaves just like File.new. However, if you associate a block with the call, open behaves differently. Instead of returning a new File object, it invokes the block, passing the newly opened File as a parameter. When the block exits, the file is automatically closed.

```
File.open("testfile", "r") do |file|
  # ... process the file
end    # << file automatically closed here
```

This second approach has an added benefit. In the earlier case, if an exception is raised while processing the file, the call to file.close may not happen. Once the file variable goes out of scope, then garbage collection will eventually close it, but this may not happen for a while. Meanwhile, resources are being held open.

This doesn't happen with the block form of File.open. If an exception is raised inside the block, the file is closed before the exception is propagated on to the caller. It's as if the open method looks like the following:

```
class File
  def File.open(*args)
    result = f = File.new(*args)
    if block_given?
      begin
        result = yield f
      ensure
        f.close
      end
    end
    return result
  end
end
```

Reading and Writing Files

The same methods that we've been using for "simple" I/O are available for all file objects. So, gets reads a line from standard input (or from any files specified on the command line when the script was invoked), and file.gets reads a line from the file object *file*.

For example, we could create a program called `copy.rb`:

```
while line = gets
  puts line
end
```

If we run this program with no arguments, it will read lines from the console and copy them back to the console. Note that each line is echoed once the Return key is pressed. (In this and later examples, we show user input in a bold font.)

```
% ruby copy.rb
These are lines
These are lines
that I am typing
that I am typing
^D
```

We can also pass in one or more filenames on the command line, in which case gets will read from each in turn:

```
% ruby copy.rb testfile
This is line one
This is line two
This is line three
And so on...
```

Finally, we can explicitly open the file and read from it:

```
File.open("testfile") do |file|
  while line = file.gets
    puts line
  end
end
```

produces:

```
This is line one
This is line two
This is line three
And so on...
```

As well as gets, I/O objects enjoy an additional set of access methods, all intended to make our lives easier.

Iterators for Reading

As well as using the usual loops to read data from an IO stream, you can also use various Ruby iterators. IO#each_byte invokes a block with the next 8-bit byte from the IO object (in this case, an object of type File). The chr method converts an integer to the corresponding ASCII character:

```
File.open("testfile") do |file|
  file.each_byte {|ch| print "#{ch.chr}:#{ch} "  }
end
```

produces:

```
T:84 h:104 i:105 s:115  :32 i:105 s:115  :32 l:108 i:105 ...
T:84 h:104 i:105 s:115  :32 i:105 s:115  :32 l:108 i:105 ...
T:84 h:104 i:105 s:115  :32 i:105 s:115  :32 l:108 i:105 ...
A:65 n:110 d:100  :32 s:115 o:111  :32 o:111 n:110 .:46 ...
```

IO#each_line calls the block with each line from the file. In the next example, we'll make the original newlines visible using String#dump so you can see that we're not cheating:

```
File.open("testfile") do |file|
  file.each_line {|line| puts "Got #{line.dump}" }
end
```

produces:

```
Got "This is line one\n"
Got "This is line two\n"
Got "This is line three\n"
Got "And so on...\n"
```

You can pass each_line any sequence of characters as a line separator, and it will break up the input accordingly, returning the line ending at the end of each line of data. That's why you see the \n characters in the output of the previous example. In the next example, we'll use the character e as the line separator:

```
File.open("testfile") do |file|
  file.each_line("e") {|line|  puts "Got #{ line.dump }" }
end
```

produces:

```
Got "This is line"
Got " one"
Got "\nThis is line"
Got " two\nThis is line"
Got " thre"
Got "e"
Got "\nAnd so on...\n"
```

If you combine the idea of an iterator with the autoclosing block feature, you get IO.foreach. This method takes the name of an I/O source, opens it for reading, calls the iterator once for every line in the file, and then closes the file automatically:

```
IO.foreach("testfile") {|line| puts line }
```

produces:

```
This is line one
This is line two
This is line three
And so on...
```

Or, if you prefer, you can retrieve an entire file into a string or into an array of lines:

```
# read into string
str = IO.read("testfile")
str.length   # =>   66
str[0, 30]   # =>   "This is line one\nThis is line "

# read into an array
arr = IO.readlines("testfile")
arr.length   # =>   4
arr[0]       # =>   "This is line one\n"
```

Don't forget that I/O is never certain in an uncertain world—exceptions will be raised on most errors, and you should be ready to rescue them and take appropriate action.

Writing to Files

So far, we've been merrily calling puts and print, passing in any old object and trusting that Ruby will do the right thing (which, of course, it does). But what exactly *is* it doing?

The answer is pretty simple. With a couple of exceptions, every object you pass to puts and print is converted to a string by calling that object's to_s method. If for some reason the to_s method doesn't return a valid string, a string is created containing the object's class name and ID, something like #<ClassName:0x123456>:

```
# Note the "w", which opens the file for writing
File.open("output.txt", "w") do |file|
  file.puts "Hello"
  file.puts "1 + 2 = #{1+2}"
end
# Now read the file in and print its contents to STDOUT
puts File.read("output.txt")
```

produces:

```
Hello
1 + 2 = 3
```

1.9 The exceptions are simple, too. The nil object will print as the empty string, and an array passed to puts will be written as if each of its elements in turn were passed separately to puts.

What if you want to write binary data and don't want Ruby messing with it? Well, normally you can simply use IO#print and pass in a string containing the bytes to be written. However, you can get at the low-level input and output routines if you really want—look at the documentation for IO#sysread and IO#syswrite on page 562.

And how do you get the binary data into a string in the first place? The three common ways are to use a literal, poke it in byte by byte, or use Array#pack:

```
str1 = "\001\002\003"    # =>   "\x01\x02\x03"
str2 = ""
str2 << 1 << 2 << 3      # =>   "\x01\x02\x03"
[ 1, 2, 3 ].pack("c*")   # =>   "\x01\x02\x03"
```

But I Miss My C++ iostream

Sometimes there's just no accounting for taste. . . . However, just as you can append an object to an Array using the << operator, you can also append an object to an output IO stream:

```
endl = "\n"
STDOUT << 99 << " red balloons" << endl
```

produces:

```
99 red balloons
```

Again, the << method uses to_s to convert its arguments to strings before sending them on their merry way.

Although we started off disparaging the poor << operator, there are actually some good reasons for using it. Because other classes (such as String and Array) also implement a << operator with similar semantics, you can quite often write code that appends to something using << without caring whether it is added to an array, a file, or a string. This kind of flexibility also makes unit testing easy. We discuss this idea in greater detail in the chapter on duck typing, starting on page 370.

Doing I/O with Strings

There are often times where you need to work with code that assumes it's reading from or writing to one or more files. But you have a problem: the data isn't in files. Perhaps it's available instead via a SOAP service, or it has been passed to you as command-line parameters. Or maybe you're running unit tests, and you don't want to alter the real file system.

Enter StringIO objects. They behave just like other I/O objects, but they read and write strings, not files. If you open a StringIO object for reading, you supply it with a string. All read operations on the StringIO object then read from this string. Similarly, when you want to write to a StringIO object, you pass it a string to be filled.

```
require 'stringio'

ip = StringIO.new("now is\nthe time\nto learn\nRuby!")
op = StringIO.new("", "w")

ip.each_line do |line|
  op.puts line.reverse
end
op.string   # =>   "\nsi won\n\nemit eht\n\nnrael ot\n!ybuR\n"
```

Talking to Networks

Ruby is fluent in most of the Internet's protocols, both low-level and high-level.

For those who enjoy groveling around at the network level, Ruby comes with a set of classes in the socket library (documented starting on page 878). These classes give you access to TCP, UDP, SOCKS, and Unix domain sockets, as well as any additional socket types supported on your architecture. The library also provides helper classes to make writing servers easier. Here's a simple program that gets information about the "mysql" user on our local machine using the finger protocol:

```
require 'socket'
client = TCPSocket.open('127.0.0.1', 'finger')
client.send("mysql\n", 0)    # 0 means standard packet
puts client.readlines
client.close
```

produces:

```
Login: _mysql                          Name: MySQL Server
Directory: /var/empty                 Shell: /usr/bin/false
Never logged in.
No Mail.
No Plan.
```

At a higher level, the `lib/net` set of library modules provides handlers for a set of application-level protocols (currently FTP, HTTP, POP, SMTP, and telnet). These are documented starting on page 773. For example, the following program lists the images that are displayed on this book's home page:

Download **samples/tutio_18.rb**

```
require 'net/http'
h = Net::HTTP.new('www.pragprog.com', 80)
response = h.get('/titles/ruby3/programming-ruby-3')
if response.message == "OK"
  puts response.body.scan(/<img alt=".*?" src="(.*?)"/m).uniq
end
```

produces:

```
http://assets1.pragprog.com/images/logo.gif?1239424264
http://assets0.pragprog.com/images/login-button.gif?1239424264
http://assets1.pragprog.com/images/covers/190x228/betas/ruby3.jpg?1236205316
http://assets1.pragprog.com/images/covers/40x48/fr_rr.jpg?1184184147
...
```

Although attractively simple, this example could be improved significantly. In particular, it doesn't do much in the way of error handling. It should really report "Not Found" errors (the infamous 404) and should handle redirects (which happen when a web server gives the client an alternative address for the requested page).

We can take this to a higher level still. By bringing the open-uri library into a program, the Kernel.open method suddenly recognizes http:// and ftp:// URLs in the filename. Not just that—it also handles redirects automatically.

Download **samples/tutio_19.rb**

```
require 'open-uri'
open('http://pragprog.com') do |f|
  puts f.read.scan(/<img alt=".*?" src="(.*?)"/m).uniq
end
```

produces:

```
http://assets1.pragprog.com/images/logo.gif?1239424264
http://assets0.pragprog.com/images/login-button.gif?1239424264
http://assets1.pragprog.com/images/front_page.png?1239424264
http://assets3.pragprog.com/images/covers/75x90/ltp2.jpg?1236205271
http://assets0.pragprog.com/images/covers/75x90/jrport.jpg?1236205229
...
```