CHAPTER 4

Developing a Basic Ruby Application

Up to this point we've focused on covering the basics of the Ruby language and looking at how it works at the ground level. In this chapter we'll move into the world of real software development and develop a full, though basic, Ruby application with a basic set of features. Once we've developed and tested the basic application, we'll look at different ways to extend it to become more useful. On our way we'll cover some new facets of development that haven't been mentioned so far in this book.

First, we're going to look at the basics of source code organization before moving on to actual programming.

Working with Source Code Files

So far in this book we've focused on using the irb immediate Ruby prompt to learn about the language. However, for developing anything you wish to reuse over and over, it's essential to store the source code in a file that can be stored on disk (or sent over the Internet, kept on CD, and so forth).

The mechanism by which you create and manipulate source code files on your system varies by operating system and personal preference. On Windows, you might be familiar with the included Notepad software for creating and editing text files. At a Linux prompt, you might be using vi, Emacs, or pico/nano. Mac users have TextEdit at their disposal. Whatever you use, you need to be able to create new files and save them as plain text so that Ruby can use them properly. In the next few sections, you're going to look at some specific tools available on each platform that tie in well with Ruby development.

Creating a Test File

The first step to developing a Ruby application is to get familiar with your text editor. Here's some guidance for each major platform.

If you're already familiar with text editors and how they relate to writing and saving source code, skip down to the section entitled "The Test Source Code File."

Windows

If you followed the instructions in Chapter 1 for downloading and installing Ruby, you'll have two text editors called SciTE and FreeRIDE in the Ruby program group in your "Start" menu. SciTE is a generic source code editing tool, whereas FreeRIDE is a Ruby-specific source code editor, written in Ruby itself. SciTE is a little faster, but FreeRIDE is more than fast enough for general development work and has better integration with Ruby.

Once you load an editor, you're presented with a blank document where you can begin to type Ruby source code (on FreeRIDE you need to use the "File" menu to create a new document). By using the "File" menu, you can also save your source code to the hard drive, as you'll do in the next section. With FreeRIDE, it's also possible to organize multiple files into a single *project*.

Mac OS X

Mac OS X has a number of text editors available. TextMate by MacroMates (`http://www.macromates.com/`), as shown in Figure 4-1, tends to be the most respected in the Ruby community, but it's not free and costs approximately $50. Xcode, included with the OS X Development Tools, is also a viable alternative, but requires that you know how to install and use the development tools (these come on your OS X installation disc). Xcode can also feel quite slow, depending on the specification of your Mac.

Included with OS X for free, however, is TextEdit. You can load TextEdit by going to your `Applications` folder and double-clicking the TextEdit icon. In its default mode, TextEdit isn't a plain text editor, but if you go to the "Format" menu and select "Make Plain Text," you'll be taken to a plain text editing mode that's suitable for editing Ruby source code.

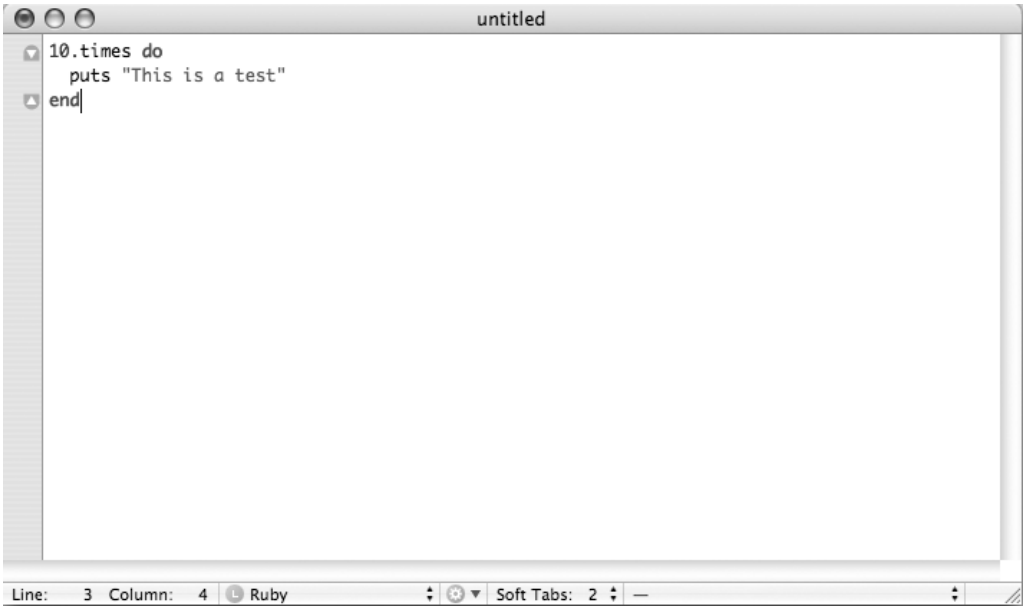

Figure 4-1. *Using TextMate*

At this point you can simply type or paste Ruby code and use the "File" ➤ "Save" menu option to save your text to a location on your drive. It would probably be good to create a folder called `ruby` within your home folder (the folder on the left that has your username in it) and save your initial Ruby source code there, as this is what the instructions assume in the next section.

Linux

Linux distributions often come with varying text editors, but there will be at least one available. If you're working entirely from the shell or terminal, you might be familiar with vi, Emacs, pico, or nano, and all of these are suitable for editing Ruby source code. If you're using Linux with a graphical interface, you might have Kate (KDE Advanced Text Editor) and/or gedit (GNOME Editor) available. All the preceding are great text and source code editors.

You could also download and install FreeRIDE, a cross-platform source code editor that's specifically designed for Ruby developers. It allows you to run your code with a single click directly from the editor (if you're using the X graphical user interface), and colors in your code in a way that reflects its syntax, which makes it easier to read. You can learn more about FreeRIDE at `http://freeride.rubyforge.org/`.

At this stage it would be a good idea to create a folder in your home directory called `ruby`, so that you can save your Ruby code there and have it in an easily remembered place.

The Test Source Code File

Once you've got an environment where you can edit and save text files, enter the following code:

```
x = 2
print "This application is running okay if 2 + 2 = #{x + x}"
```

■ **Note** If this code looks like nonsense to you, you've skipped too many chapters. Head back to Chapter 3! This chapter requires full knowledge of everything covered in Chapter 3.

Save the code with a filename of `a.rb` in a folder or directory of your choice. It's advisable that you create a folder called `ruby` located somewhere that's easy to find. On Windows this might be directly off of your C drive, and on OS X or Linux this could be a folder located in your home directory.

■ **Note** RB is the de facto standard file extension for Ruby files, much like PHP is standard for PHP, TXT is common for text files, and JPG is standard for JPEG images.

Now you're ready to run the code.

Running Your Source Code

Once you've created the basic Ruby source code file, `a.rb`, you need to get Ruby to execute it. As always, the process by which to do this varies by operating system. Read the particular following section that matches your operating system. If your operating system isn't listed, the OS X and Linux instructions are most likely to match those for your platform.

Whenever this book asks you to "run" your program, this is what you'll be doing each time.

■ **Note** Even though you're going to be developing an application in this chapter, there are still times when you'll want to use irb to follow along with the tests or basic theory work throughout the chapter. Use your judgment to jump between these two methods of development. irb is extremely useful for testing small concepts and short blocks of code without the overhead of jumping back and forth between a text editor and the Ruby interpreter.

Windows

If you're using the SciTE or FreeRIDE programs that came with the Ruby installer for Windows, you can run Ruby programs directly from them (see Figure 4-2). In both programs you can press the F5 function key to run your Ruby code. Alternatively you can use the menus ("Tools" ➤ "Go" in SciTe, and "Run" ➤ "Run" in FreeRIDE). However, before you do this, it's important to make sure you have saved your Ruby code. If not, the results might be unpredictable (running old code from a prior save, for example) or you'll be prompted to save your work.

If running the `a.rb` code gives a satisfactory output in the output view pane (to the right on SciTE, and at the bottom on FreeRIDE), you're ready to move on to the section, "Our Application: A Text Analyzer."

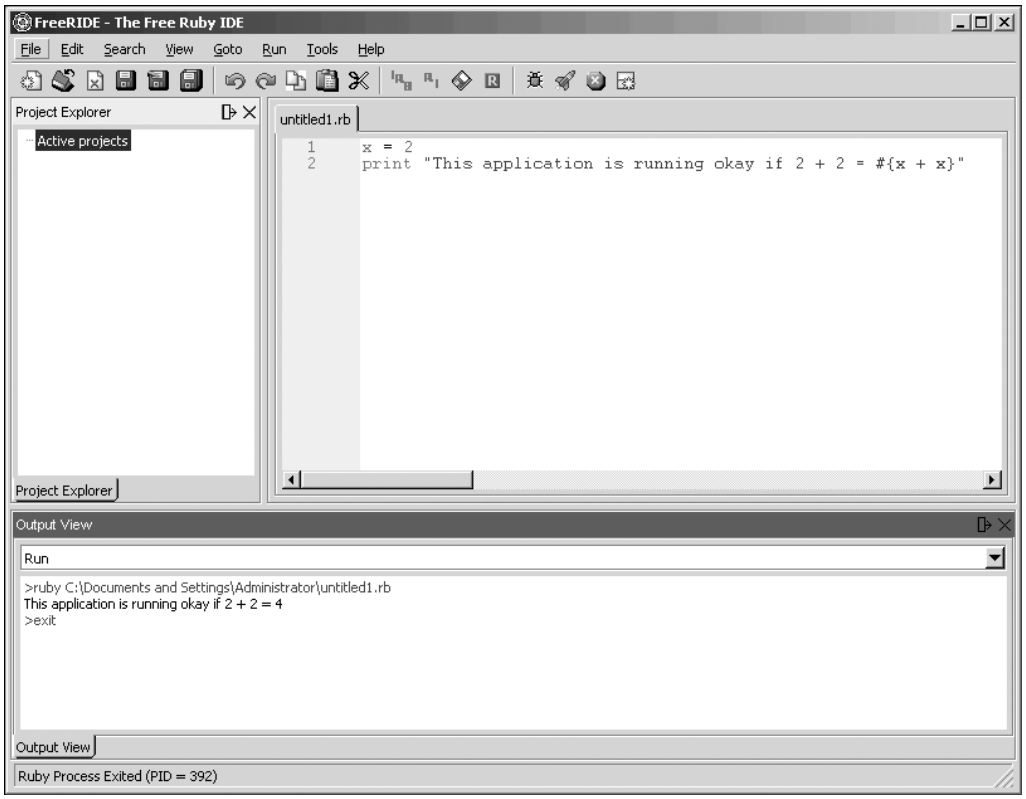

Figure 4-2. *Running code in FreeRIDE on Microsoft Windows (notice the output in the bottom pane)*

Alternatively, you might prefer to run Ruby from the command prompt. To do this, load up the command prompt ("Start" menu ➤ "Run" ➤ Type **cmd** and click "OK"), navigate to the folder containing `a.rb` using the `cd` command, and then type **ruby a.rb**.

However, this method is only advised if you understand how to navigate your hard drive from the command prompt. Another option, if you're comfortable with creating shortcuts, is to create a shortcut to the Ruby executable file (`ruby.exe`) and drop your source code file(s) onto it.

Mac OS X

The simplest method to run Ruby applications on Mac OS X is from the Terminal, much in the same way as irb is run. The Terminal was explained in Chapter 1. If you followed the preceding instructions, continue like so:

1. Launch the Terminal (found in `Applications/Utilities`).
2. Use `cd` to navigate to the folder where you placed `a.rb` like so: `cd ~/ruby` . This tells the Terminal to take you to the `ruby` folder located off of your home user folder.
3. Type **ruby a.rb** and press Enter to execute the `a.rb` Ruby script.
4. If you get an error such as `ruby: No such file or directory -- a.rb (LoadError)`, you aren't in the same folder as the `a.rb` source file and need to establish where you have saved it.

If you get a satisfactory response from `a.rb`, you're ready to move on to the section, "Our Application: A Text Analyzer."

Linux and Other Unix-Based Systems

In Linux or other Unix-based systems, you run your Ruby applications from the shell (that is, within a terminal window) in the same way that you ran irb. The process to run irb was explained in Chapter 1, so if you've forgotten how to get that far, you need to recap yourself before continuing like so:

1. Launch your terminal emulator (xterm, for example) so you get a Linux shell/command prompt.
2. Navigate to the directory where you placed `a.rb` using the `cd` command (for example, `cd ~/ruby` takes you to the `ruby` directory located directly under your home directory, usually `/home/yourusernamehere/`).
3. Type **ruby a.rb** and press Enter to make Ruby execute the `a.rb` script.

If you get a satisfactory response from `a.rb`, you're ready to move on.

TEXT EDITORS VS. SOURCE CODE EDITORS

Previously I've stated that source code is basically the same as plain text. This is true, and although you can write your code in a general text editor, some benefits can be obtained by using a specialist source code editor (or a development IDE—Integrated Development Environment).

The FreeRIDE editor is an example of an editor specifically created for Ruby developers. It edits text, as with any other text editor, but offers extended features such as source code highlighting and the ability to run code directly from the editor.

Some developers find source code syntax highlighting an invaluable feature, as it makes their code easier to read. Variable names, expressions, string literals, and other elements of your source code are all given different colors, which makes it easy to pick them out.

Whether you choose a source code editor or a basic text editor depends on your own preference, but it's worth trying both. Many developers prefer the freedom of a regular text editor and then running their Ruby programs from the command line, whereas others prefer to work entirely within a single environment.

FreeRIDE is available from `http://freeride.rubyforge.org/`, and a competing source code editor for Ruby *and* Rails, called RadRails, is available at `http://www.radrails.org/`. It's certainly worth investigating these other editors on your platform in case they fit in more with how you wish to work.

Our Application: A Text Analyzer

The application you're going to develop in this chapter will be a text analyzer. Your Ruby code will read in text supplied in a separate file, analyze it for various patterns and statistics, and print out the results for the user. It's not a 3D graphical adventure or a fancy Web site, but text processing programs are the bread and butter of systems administration and most application development. They can be vital for parsing log files and user-submitted text on Web sites, and manipulating other textual data.

Ruby is well suited for text and document analysis with its regular expression features, along with the ease of use of `scan` and `split`, and you'll be using these heavily in your application.

■ **Note** With this application you'll be focusing on implementing the features quickly, rather than developing an elaborate object-oriented structure, any documentation, or a testing methodology. I'll be covering object orientation and its usage in larger programs in depth in Chapter 6, and documentation and testing are covered in Chapter 8.

Required Basic Features

Your text analyzer will provide the following basic statistics:

- Character count
- Character count (excluding spaces)
- Line count
- Word count
- Sentence count
- Paragraph count
- Average number of words per sentence
- Average number of sentences per paragraph

In the last two cases, the statistics are easily calculated from each other. That is, once you have the total number of words and the total number of sentences, it becomes a matter of a simple division to work out the average number of words per sentence.

Building the Basic Application

When starting to develop a new program, it's useful to think of the key steps involved. In the past it was common to draw *flow charts* to show how the operation of a computer program would flow, but it's easy to experiment, to change things about, and to remain agile with modern tools, such as Ruby. Let's outline the basic steps as follows:

1. Load in a file containing the text or document you want to analyze.
2. As you load the file line by line, keep a count of how many lines there were (one of your statistics taken care of).
3. Put the text into a string and measure its length to get your character count.
4. Temporarily remove all whitespace and measure the length of the resulting string to get the character count excluding spaces.
5. Split out all the whitespace to find out how many words there are.
6. Split on full stops to find out how many sentences there are.
7. Split on double newlines to find out how many paragraphs there are.
8. Perform calculations to work out the averages.

Create a new, blank Ruby source file and save it as `analyzer.rb` in your Ruby folder. As you work through the next few sections you'll be able to fill it out.

Obtaining Some Dummy Text

Before you start to code, the first step is to get some test data that your analyzer can process. The first chapter of *Oliver Twist* is an ideal piece of text to use, as it's copyright free and easy to obtain. It's also of a reasonable length. You can find the text at `http://www.rubyinside.com/book/oliver.txt` or `http://www.dickens-literature.com/Oliver_Twist/0.html` for you to copy into a local text file. Save the file in the same folder as where you saved `a.rb` and call it `text.txt`. Your application will read from `text.txt` by default (although you'll make it be more dynamic and able to accept other sources of data later on).

■ **Tip** If the preceding Web pages are unavailable at the time of reading, use your favorite search engine to search for "twist workhouse rendered profound thingummy" and you're guaranteed to find it. Alternatively, use any large block of text you can obtain.

If you're using the *Oliver Twist* text and want your results to match up roughly with those given as examples throughout this chapter, make sure you only copy and paste the text including and between these sections:

```
Among other public buildings in a certain town, which for many
reasons it will be prudent to refrain from mentioning
```

And:

```
Oliver cried lustily. If he could have known that he was an
orphan, left to the tender mercies of church-wardens and
overseers, perhaps he would have cried the louder.
```

Loading Text Files and Counting Lines

Now it's time to get coding! The first step is to load the file. Ruby provides a comprehensive set of file manipulation methods via the `File` class. Whereas other languages can make you jump through hoops to work with files, Ruby keeps the interface simple. Here's some code that opens up your `text.txt` file:

```
File.open("text.txt").each { |line| puts line }
```

Type this into `analyzer.rb` and run the code. If `text.txt` is in the current directory, the result is that you'll see the entire text file flying up the screen.

You're asking the `File` class to open up `text.txt`, and then, much like with an array, you can call the `each` method on the file directly, resulting in each line being passed to the inner code block one by one, where `puts` sends the line as output to the screen. (In Chapter 9 you'll look at how file access and manipulation work in more detail, along with better techniques than are used in this chapter!)

Edit the code to look like this instead:

```
line_count = 0
File.open("text.txt").each { |line| line_count += 1 }
puts line_count
```

You initialize `line_count` to store the line count, then open the file and iterate over each line, while incrementing `line_count` by `1` each time. When you're done, you print the total to the screen (approximately 121 if you're using the *Oliver Twist* chapter). You have your first statistic!

You've counted the lines, but still don't have access to the contents of the file to count the words, paragraphs, sentences, and so forth. This is easy to fix. Let's change the code a little, and add a variable, `text`, to collect the lines together as one as we go:

```
text=''
line_count = 0
File.open("text.txt").each do |line|
  line_count += 1
  text << line
end

puts "#{line_count} lines"
```

■ **Note** Remember that using { and } to surround blocks is the standard style for single line blocks, but using `do` and `end` is preferable for multiline blocks. However, this is a convention rather than a requirement.

Compared to your previous attempt, this code introduces the `text` variable and adds each line onto the end of it in turn. When the iteration over the file has finished—that is, when you run out of lines—`text` contains the entire file in a single string ready for you to use.

That's a simple-looking way to get the file into a single string and count the lines, but `File` also has other methods that can be used to read files more quickly. For example, you can rewrite the preceding code like this:

```
lines = File.readlines("text.txt")
line_count = lines.size
text = lines.join

puts "#{line_count} lines"
```

Much simpler! `File` implements a `readlines` method that reads an entire file into an array, line by line. You can use this both to count the lines and join them all into a single string.

Counting Characters

The second easiest statistic to work out is the number of characters in the file. As you've collected the entire file into the `text` variable, and `text` is a string, you can use the `length` method that all strings supply to get the exact size of the file, and therefore the number of characters.

To the end of the previous code in `analyzer.rb`, add the following:

```
total_characters = text.length
puts "#{total_characters} characters"
```

If you ran `analyzer.rb` now with the *Oliver Twist* text, you'd get output like this:

```
121 lines
6165 characters
```

The second statistic you wanted to get relating to characters was a character total excluding whitespace. If you can remember back to Chapter 3, strings have a `gsub` method that performs a global substitution (like a search and replace) upon the string. For example:

```
"this is a test".gsub(/t/, 'X')
```

```
Xhis is a XesX
```

You can use `gsub` to eradicate the spaces from your `text` string in the same way, and then use the `length` method to get the length of the newly "de-spacified" `text`. Add the following code to `analyzer.rb`:

```
total_characters_nospaces = text.gsub(/\s+/, '').length
puts "#{total_characters_nospaces} characters excluding spaces"
```

If you run `analyzer.rb` in its current state against the *Oliver Twist* text, the results should be similar to the following:

```
121 lines
6165 characters
5055 characters (excluding spaces)
```

Counting Words

A common feature offered by word processing software is a "word counter." All it does is count the number of complete words in your document or a selected area of text. This information is useful to work out how many pages the document will take up when printed. Many assignments also have requirements for a certain number of words, so knowing the number of words in a piece of text is certainly useful.

You can approach this feature in a couple of ways:

1. Count the number of groups of contiguous letters using `scan`.
2. Split the text on whitespace and count the resulting fragments using `split` and `size`.

Let's look at each method in turn to see what's best. Recall from Chapter 3 that `scan` works by iterating over a string of text and finding certain patterns over and over. For example:

```
puts "this is a test".scan(/\w/).join
```

```
thisisatest
```

In this example, `scan` looked through the string for anything matching `\w`, a special term representing all alphanumeric characters (and underscores), and placed them into an array that you've joined together into a string and printed to the screen.

You can do the same with groups of alphanumeric characters. In Chapter 3 you learned that to match multiple characters with a regular expression, you could follow the character with `+`. So let's try again:

```
puts "this is a test".scan(/\w+/).join('-')
```

```
this-is-a-test
```

This time, `scan` has looked for all *groups* of alphanumeric characters and placed them into the array that you've then joined together into a string using - as the separation character.

To get the number of words in the string, you can use the `length` or `size` array methods to count the number of elements rather than join them together:

```
puts "this is a test".scan(/\w+/).length
```

```
4
```

Excellent! So what about the `split` approach?

The `split` approach demonstrates a core tenet of Ruby (as well as some other languages, particularly Perl) that "There's always more than one way to do it!" Analyzing different methods to solve the same problem is a crucial part of becoming a good programmer, as different methods can vary in their efficacy.

Let's split the string by spaces and get the length of the resulting array, like so:

```
puts "this is a test".split.length
```

```
4
```

As it happens, by *default* `split` will split by whitespace (single or multiple characters of spaces, tabs, newlines, and so on), and that makes this code shorter and easier to read than the `scan` alternative.

So what's the difference between these two methods? Simply, one is looking for words and returning them to you for you to count, and the other is splitting the string by that which separates words—whitespace—and telling you how many parts the string was broken into. Interestingly, these two approaches can yield different results:

```
text = "First-class decisions require clear-headed thinking."
puts "Scan method: #{text.scan(/\w+/).length}"
puts "Split method: #{text.split.length}"
```

```
Scan method: 7
Split method: 5
```

Interesting! The `scan` method is looking through for all blocks of alphanumeric characters, and, sure enough, there are seven in the sentence. However, if you split by spaces, there are five words. The reason is the hyphenated words. Hyphens aren't "alphanumeric," so `scan` is seeing "first" and "class" as separate words.

Returning to `analyzer.rb`, let's apply what we've learned here. Add the following:

```
word_count = text.split.length
puts "#{word_count} words"
```

Running the complete `analyzer.rb` gets these results:

```
122 lines
6166 characters
5055 characters (excluding spaces)
1093 words
```

Counting Sentences and Paragraphs

Once you understand the logic of counting words, counting the sentences and paragraphs becomes easy. Rather than splitting on whitespace, sentences and paragraphs have different splitting criteria.

Sentences end with full stops, question marks, and exclamation marks. They can also be separated with dashes and other punctuation, but we won't worry about these rare cases here. The split is simple. Instead of asking Ruby to split the text on one type of character, you simply ask it to split on any of three types of characters, like so:

```
sentence_count = text.split(/\.|\?|!/).length
```

The regular expression looks odd here, but the full stop, question mark, and exclamation mark are clearly visible. Let's look at the regular expression directly:

```
/\.|\?|!/
```

The forward slashes at the start and the end are the usual delimiters for a regular expression, so those can be ignored. The first section is `\.` and this represents a full stop.

The reason why you can't just use `.` without the backslash in front is because `.` represents "any character" in a regular expression (as covered in Chapter 3), so it needs to be *escaped* with the backslash to identify itself as a literal full stop. This also explains why the question mark is escaped with a backslash, as a question mark in a regular expression usually means "zero or one of the previous character"—also covered in Chapter 3. The `!` is not escaped, as it has no other meaning in terms of regular expressions.

The pipes (`|` characters) separate the three main characters, which means they're treated separately so that `split` can match one or another of them. This is what allows the split to split on periods, question marks, *and* exclamation marks all at the same time. You can test it like so:

```
puts "Test code! It works. Does it? Yes.".split(/\.|\?|!/).length
```

```
4
```

Paragraphs can also be split apart with regular expressions. Whereas paragraphs in a printed book, such as this one, tend not to have any spacing between them, paragraphs that are typed on a computer typically do, so you can split by a double newline (as represented by the special combination `\n\n`—simply, two newlines in succession) to get the paragraphs separated. For example:

```
text = %q{
This is a test of
paragraph one.

This is a test of
paragraph two.

This is a test of
paragraph three.
}

puts text.split(/\n\n/).length
```

```
3
```

Let's add both these concepts to `analyzer.rb`:

```
paragraph_count = text.split(/\n\n/).length
puts "#{paragraph_count} paragraphs"

sentence_count = text.split(/\.|\?|!/).length
puts "#{sentence_count} sentences"
```

Calculating Averages

The final statistics required for your basic application are the average number of words per sentence, and the average number of sentences per paragraph. You already have the paragraph, sentence, and word counts available in the variables `word_count`, `paragraph_count`, and `sentence_count`, so only basic arithmetic is required, like so:

```
puts "#{sentence_count / paragraph_count} sentences per paragraph (average)"
puts "#{word_count / sentence_count} words per sentence (average)"
```

The calculations are so simple that they can be interpolated directly into the output commands rather than precalculated.

The Source Code So Far

You've been updating the source code as you've gone along, and in each case you've put the logic next to the `puts` statement that shows the result to the user. However, for the final version of your basic application, it'd be tidier to separate the logic from the presentation a little and put the calculations in a separate block of code before everything is printed to the screen.

There are no logic changes, but the finished source for `analyzer.rb` looks a little cleaner this way:

```
lines = File.readlines("text.txt")
line_count = lines.size
text = lines.join
word_count = text.split.length
character_count = text.length
character_count_nospaces = text.gsub(/\s+/, '').length
paragraph_count = text.split(/\n\n/).length
sentence_count = text.split(/\.|\?|!/).length

puts "#{line_count} lines"
puts "#{character_count} characters"
```

```
puts "#{character_count_nospaces} characters excluding spaces"
puts "#{word_count} words"
puts "#{paragraph_count} paragraphs"
puts "#{sentence_count} sentences"
puts "#{sentence_count / paragraph_count} sentences per paragraph (average)"
puts "#{word_count / sentence_count} words per sentence (average)"
```

If you've made it this far and everything's making sense, congratulations are due. Let's look at how to extend our application a little further with some more interesting statistics.

Adding Extra Features

Your analyzer has a few basic functions, but it's not particularly interesting. Line, paragraph, and word counts are useful statistics, but with the power of Ruby you can extract significantly more interesting data from the text. The only limit is your imagination, but in this section you'll look at a couple other features you can implement, and how to do so.

■ **Note** When developing software it's always worth considering the likelihood of the software being extended or tweaked in the future and planning ahead for the possibility. Many development bottlenecks have occurred when systems were designed too rigidly to cope with changing circumstances!

Percentage of "Useful" Words

Most written material, including this very book, contains a high number of words that, although providing context and structure, are not directly useful or interesting. In the last sentence, the words "that," "and," "are," and "or" are not of particular interest, even if the sentence would make less sense without them.

These words are typically called "stop words," and are often ignored by computer systems whose job is to analyze and search through text, because they aren't words most people are likely to be searching for (as opposed to nouns, for example). Google is a perfect example of this, as it doesn't want to have to store information that takes up space and that's generally irrelevant to searches.

■ **Note** For more information about stop words, including links to complete lists, visit `http://en.wikipedia.org/wiki/Stop_words`.

It could be assumed that more "interesting" text, or text by a more proficient author, might have a lower percentage of stop words and a higher percentage of useful or interesting words. You can easily extend your application to work out the percentage of non–stop words in the supplied text.

The first step is to build up a list of stop words. There are hundreds of possible stop words, but you'll start with just a handful. Let's create an array to hold them:

```
stop_words = %w{the a by on for of are with just but and to the my I has some in}
```

This code results in an array of stop words being assigned to the `stop_words` variable.

■ **Tip** In Chapter 3, you saw arrays being defined like so: `x = ['a', 'b', 'c']`. However, like many languages, Ruby has a shortcut that builds arrays quickly with string-separated text. This segment can be shorted to the equivalent `x = %w{a b c}`, as demonstrated in the preceding stop word code.

For demonstration purposes, let's write a small, separate program to test the concept:

```
text = %q{Los Angeles has some of the nicest weather in the country.}
stop_words = %w{the a by on for of are with just but and to the my I has some}

words = text.scan(/\w+/)
key_words = words.select { |word| !stop_words.include?(word) }

puts key_words.join(' ')
```

When you run this code, you get the following result:

```
Los Angeles nicest weather country
```

Cool, right? First you put some text into the program, then the list of stop words. Next you get all the words from `text` into an array called `words`. Then you get to the magic:

```
key_words = words.select { |word| !stop_words.include?(word) }
```

This line first takes your array of words, `words`, and calls the `select` method with a block of code to process for each word (like the iterators you played with in Chapter 3). The `select` method is a method available to all arrays and hashes that returns the elements of that array or hash that match the expression in the code block.

In this case, the code in the code block takes each word via the variable `word`, and asks the `stop_words` array whether it includes any elements equal to `word`. This is what `stop_words.include?(word)` does.

The exclamation mark (`!`) before the expression negates the expression (an exclamation mark negates any Ruby expression). The reason for this is you *don't* want to select words that *are* in the `stop_words` array. You want to select words that *aren't*.

In closing, then, you `select` all elements of `words` that are *not* included in the `stop_words` array and assign them to `key_words`. Don't read on until that makes sense, as this type of single-line construction is common in Ruby programming.

After that, working out the percentage of non–stop words to all words uses some basic arithmetic:

```
((key_words.length.to_f / words.length.to_f) * 100).to_i
```

The reason for the `.to_f`'s is so that the lengths are treated as floating decimal point numbers, and the percentage is worked out more accurately. When you work it up to the real percentage (out of 100), you can convert back to an integer once again.

You'll see how this all comes together in the final version at the end of this chapter.

Summarizing by Finding "Interesting" Sentences

Word processors such as Microsoft Word generally have summarization features that can take a long piece of text and seemingly pick out the best sentences to produce an "at-a-glance" summary. The mechanisms for producing summaries have become more complex over the years, but one of the simplest ways to develop a summarizer of your own is to scan for sentences with certain characteristics.

One technique is to look for sentences that are of about average length and that look like they contain nouns. Tiny sentences are unlikely to contain anything useful, and long sentences are likely to be simply too long for a summary. Finding nouns reliably would require systems that are far beyond the scope of this book, so you could "cheat" by looking for words that indicate the presence of useful nouns in the same sentence, such as "is" and "are" (for example, "Noun is," "Nouns are," "There are *x* nouns").

Let's assume that you want to throw away two-thirds of the sentences—a third that are the shortest sentences and a third that are the longest sentences—leaving you with an ideal third of the original sentences that are ideally sized for your task.

For ease of development, let's create a new program from scratch, and transfer your logic over to the main application later. Create a new program called `summarize.rb` and use this code:

```
text = %q{
Ruby is a great programming language. It is object oriented
and has many groovy features. Some people don't like it, but that's
not our problem! It's easy to learn. It's great. To learn more about Ruby,
visit the official Ruby Web site today.
}

sentences = text.gsub(/\s+/, ' ').strip.split(/\.|\?|\!/)
sentences_sorted = sentences.sort_by { |sentence| sentence.length }
one_third = sentences_sorted.length / 3
ideal_sentences = sentences_sorted.slice(one_third, one_third + 1)
ideal_sentences = ideal_sentences.select { |sentence| sentence =~ /is|are/ }
puts ideal_sentences.join(". ")
```

And for good measure run it to see what happens:

```
Ruby is a great programming language. It is object oriented and has many groovy
features
```

Seems like a success! Let's walk through the program.

First, you define the variable text to hold the long string of multiple sentences, much like in `analyzer.rb`. Next you split `text` into an array of sentences like so:

```
sentences = text.gsub(/\s+/, ' ').strip.split(/\.|\?|!/)
```

This is slightly different from the method used in `analyzer.rb`. There is an extra `gsub` in the chain, as well as `strip`. The `gsub` gets rid of all large areas of whitespace and replaces them with a single space (`\s+` meaning "one or more whitespace characters"). This is simply for cosmetic reasons. The `strip` removes all extra whitespace from the start and end of the string. The `split` is then the same as that used in the analyzer.

Next you sort the sentences by their lengths, as you want to ignore the shortest third and the longest third:

```
sentences_sorted = sentences.sort_by { |sentence| sentence.length }
```

Arrays and hashes have the `sort_by` method that rearranges them into almost any order you want. `sort_by` takes a code block as its argument, where the code block is an expression that defines what to sort by. In this case, you're sorting the `sentences` array. You pass each sentence in as the `sentence` variable, and choose to sort them by their length, using the `length` method upon the sentence. After this line, `sentences_sorted` contains an array with the sentences in length order.

Next you need to get the middle third of the length-sorted sentences in `sentences_sorted`, as these are the ones you've deemed to be probably the most interesting. To do this you can divide the length of the array by `3`, to get the number of elements in a third, and then grab that number of elements from one third into the array (note that you grab one extra element to compensate for rounding caused by integer division). This is done like so:

```
one_third = sentences_sorted.length / 3
ideal_sentences = sentences_sorted.slice(one_third, one_third + 1)
```

The first line takes the length of the array and divides it by `3` to get the quantity that is equal to "a third of the array." The second line uses the `slice` method to "cut out" a section of the array to assign to `ideal_sentences`. In this case, assume that the `sentences_sorted` is 6 elements long. `6` divided by `3` is `2`, so a third of the array is 2 elements long. The `slice` method then cuts *from* element `2` *for* `2` (plus `1`) elements, so you effectively carve out elements `2`, `3`, and `4` (remember that array elements start counting from 0). This means you get the "inner third" of the ideal-lengthed sentences you wanted.

The penultimate line checks to see if the sentence includes the word "is" or "are," and only accepts each sentence if so:

```
ideal_sentences = ideal_sentences.select { |sentence| sentence =~ /is|are/ }
```

It uses the `select` method, as the "stop word" removal code in the previous section did. The expression in the code block uses a regular expression that matches against `sentence`, and only returns true if "is" or "are" are present within `sentence`. This means `ideal_sentences` now only contains sentences that are in the middle third length-wise *and* contain either "is" or "are."

The final line simply joins the `ideal_sentences` together with a full stop and space between them to make them readable:

```
puts ideal_sentences.join(". ")
```

Analyzing Files Other Than text.txt

So far your application has the filename `text.txt` hard-coded into it. This is acceptable, but it'd be a lot nicer if you could specify, when you run the program, what file you want the analyzer to process.

■ **Note** This technique is only practical to demonstrate if you're running `analyzer.rb` from a command prompt or shell, as on Mac OS X or Linux (or Windows if you're using the Windows command prompt). If you're using an IDE on Windows, this section will be read-only for you.

Typically, if you're starting a program from the command line, you can append parameters onto the end of the command and the program processes them. You can do the same with your Ruby application.

Ruby automatically places any parameters that are appended to the command line when you launch your Ruby program into a special array called `ARGV`. To test it out, create a new script called `argv.rb` and use this code:

```
puts ARGV.join('-')
```

From the command prompt, run the script like so:

```
ruby argv.rb
```

The result will be blank, but then try to run it like so:

```
ruby argv.rb test 123
```

```
test-123
```

This time the parameters are taken from `ARGV`, joined together with a hyphen, and displayed on screen. You can use this to replace the reference to `text.txt` in `analyzer.rb` by replacing `"text.txt"` with `ARGV[0]` or `ARGV.first` (which both mean exactly the same thing—the first element of the `ARGV` array). The line that reads the file becomes the following:

```
lines = File.readlines(ARGV[0])
```

To process `text.txt` now, you'd run it like so:

```
ruby analyzer.rb text.txt
```

You'll learn more about deploying programs and making them friendly to other users, along with `ARGV`, in Chapter 10.

The Completed Program

You've already got the source for the completed basic program, but it's time to add all the new, extended features from the previous few sections to `analyzer.rb` to create the final version of your text analyzer.

■ **Note** Remember that source code for this book is available in the Source Code/Download area at `http://www.apress.com`, so it isn't strictly necessary to type in code directly from the book.

Here we go:

```
# analyzer.rb -- Text Analyzer

stop_words = %w{the a by on for of are with just but and to the my I has some in}
lines = File.readlines("text.txt")
line_count = lines.size
text = lines.join

# Count the characters
character_count = text.length
character_count_nospaces = text.gsub(/\s+/, '').length

# Count the words, sentences, and paragraphs
word_count = text.split.length
sentence_count = text.split(/\.|\?|!/).length
paragraph_count = text.split(/\n\n/).length

# Make a list of words in the text that aren't stop words,
# count them, and work out the percentage of non-stop words
# against all words
all_words = text.scan(/\w+/)
good_words = all_words.select{ |word| !stop_words.include?(word) }
good_percentage = ((good_words.length.to_f / all_words.length.to_f) * 100).to_i

# Summarize the text by cherry picking some choice sentences
sentences = text.gsub(/\s+/, ' ').strip.split(/\.|\?|\!/)
sentences_sorted = sentences.sort_by { |sentence| sentence.length }
one_third = sentences_sorted.length / 3
ideal_sentences = sentences_sorted.slice(one_third, one_third + 1)
ideal_sentences = ideal_sentences.select { |sentence| sentence =~ /is|are/ }

# Give the analysis back to the user
puts "#{line_count} lines"
puts "#{character_count} characters"
puts "#{character_count_nospaces} characters (excluding spaces)"
puts "#{word_count} words"
```

```
puts "#{sentence_count} sentences"
puts "#{paragraph_count} paragraphs"
puts "#{sentence_count / paragraph_count} sentences per paragraph (average)"
puts "#{word_count / sentence_count} words per sentence (average)"
puts "#{good_percentage}% of words are non-fluff words"
puts "Summary:\n\n" + ideal_sentences.join(". ")
puts "-- End of analysis"
```

■ **Note** If you're a Windows user, you might want to replace the `ARGV[0]` reference with an explicit reference to `"text.txt"` to make sure it works okay from FreeRIDE or SciTE. However, if you're running the program from the command prompt, it should operate correctly.

Running the completed `analyzer.rb` with the *Oliver Twist* text now results in an output like so:

```
121 lines
6165 characters
5055 characters (excluding spaces)
1093 words
18 paragraphs
45 sentences
2 sentences per paragraph (average)
24 words per sentence (average)
76% of words are non-fluff words
Summary:

' The surgeon leaned over the body, and raised the left hand.  Think what it is
to be a mother, there's a dear young lamb do.  'The old story,' he said, shaking
his head: 'no wedding-ring, I see.  What an excellent example of the power of
dress, young Oliver Twist was. ' Apparently this consolatory perspective of a
mother's prospects failed in producing its due effect. ' The surgeon had been
sitting with his face turned towards the fire: giving the palms of his hands a
warm and a rub alternately. ' 'You needn't mind sending up to me, if the child
cries, nurse,' said the surgeon, putting on his gloves with great deliberation.
She had walked some distance, for her shoes were worn to pieces; but where
she came from, or where she was going to, nobody knows. ' He put on his hat,
and, pausing by the bed-side on his way to the door, added, 'She was a
good-looking girl, too; where did she come from
-- End of analysis
```

Try `analyzer.rb` with some other text of your choice (a Web page, perhaps) and see if you can make improvements to its features. This application is ripe for improvement with the concepts you'll learn over the next several chapters, so keep it in mind if you're looking for some code to play with.

CODE COMMENTS

You might notice text in source code prefixed with # symbols. These are *comments* and are generally used in programs for the benefit of the original developer(s), along with anyone else that might need to read the source code. They're particularly useful for making notes to remind you of why you took a particular course of action that you're likely to forget in future.

You can place comments in any Ruby source code file on their own lines, or even at the end of a line of code. Here are some valid examples of commenting in Ruby:

```
puts "2+2 = #{2+2}" # Adds 2+2 to make 4
# A comment on a line by itself
```

As long as a comment is on a line by itself, or is the last thing on a line, it's fine. Comment liberally, and your code will be easier to understand.

Summary

In this chapter you developed a complete, basic application that had a set of requirements and desired features. You then extended it with some nonessential, but useful, elaborations. Ruby makes developing quick applications a snap.

The application you've developed in this chapter has demonstrated that if you have a lot of text to process or a number of calculations to do, and you're dreading doing the work manually, Ruby can take the strain.

Chapter 4 marks the end of the practical programming exercises in the first part of this book. Next, in Chapter 5, you'll take a look at the history of Ruby; Ruby's community of developers; the historical reasons behind certain features in Ruby; and learn how to get help from, and become part of, the Ruby community. Code makes up only half the journey to becoming a great programmer!