Ruby Fundamentals - Structure and Execution of Ruby Programs

April 16, 2025

1 Introduction

This chapter explains the structure of Ruby programs. It starts with the lexical structure, covering tokens and the characters that comprise them. Next, it covers the syntactic structure of a Ruby program, explaining how expressions, control structures, methods, classes, and son are written as a series of tokens. Finally, the chapter describes files of Ruby code,. explaining how Ruby programs can be split across multiple files and how the Ruby interpreter executes a file of Ruby code.

2 Lexical Structure

The Ruby interpreter parses a program as a sequence of *tokens*. Tokens include comments, literals, punctuation, identifiers, and keywords. This section introduces these types of tokens and also includes important information about the characters that comprise the tokens and the whitespace that separates the tokens.

2.1 Comments

Comments in Ruby begin with a `#` character and continue to the end of line. The Ruby interpreter ignores the `#` character and any text that follows it (but does not ignore the newline character, which is meaningful whitespace and may serve as a statement terminator). If a `#` character appears within a string or regular expression literal, then it is simply part of the string or regular expression and does not introduce a comment:

```
# This entire line is a comment
x = "# This is a string" # And this is a comment
y = /#This is a regular expression/ # Here's another comment
```

Multiline comments are usually written simply by beginning each line with a separate # character. Note that Ruby has no equivalent of the C-style `/*...*/` comment. There is no way to embed a comment in the middle of a line of code.

2.1.1 Embedded documents

Ruby supports another style of multiline comment known as an *embedded document*. These start on a line that begins =begin and continue until (and include) a line that begins with =end. Any text that appears after =begin or =end is part of the comment and is also ignored, but that extra text must be separated from the =begin and =end at least one space.

Embedded documents are convenient way to comment out long blocks of code without prefixing each line with a # character:

```
=begin Someone needs to fix the broken code below!
 # content of block of code
=end
```

As their name implies, embedded documents can be used to include long blocks of documentation within a program, or to embed source code of another language within a Ruby program. Embedded documents are usually intended to be used by some kind of postprocessing tool that is run over the Ruby source code, and it is typical to follow =begin with an identifier that indicates which tool the comment is intended for.

2.1.2 Documentation comments

Ruby programs can include embedded API documentation as specially formatted comments that precede method, class, and module definitions. You can browse this documentation using the *ri* tool. The *rdoc* tool extracts documentation comments from Ruby source and formats them as HTML or prepares them for display by *ri*.

Documentation comments must come immediately before che module, class, or method whose API they document. They are usually written as multiline comments where each line begins with #, but they can also be written as embedded documents what start =begin rdoc.

2.2 Literals

Literals are values that appear directly in Ruby source code. They include numbers, strings of text, and a regular expressions. Ruby number and string literal syntax is actually quite complicated.

Punctuation	
$	Global variables are prefixed with a d
	Inst
?	As a he
!	Method names may end with an exlamation point to indicate that they should be used cau
=	Methods whose names end with

Table 1: Punctuations and their meaning

2.3 Punctuation

Ruby uses punctuation characters for a number of purposes. Most Ruby operators are written using punctuation characters, such as `+` for addition, `*` for multiplication, and `||` for the boolean OR operator. Punctuation characters also serve to delimit string, regular expression, array, and hash literals, and to group and separate expressions, method arguments, and array indexes. We'll see miscellaneous other uses of punctuation scattered throughout Ruby syntax.

2.4 Identifiers

An *identifier* is simply a name. Ruby uses identifiers to name variables, methods, classes, and so forth. Ruby identifiers consist of letters, numbers, and underscore characters, but they may not begin with a number. IOdentifiers may not include whitespace or nonprinting characters, and they may not include punctuation characters except as described here.

Identifiers that begin with a capital lettera A-Z are constants, and the Ruby interpreter will issue a warning if you alter the value of such an identifier. Class and module names must begin with initial capital letters.

By convention, multiword identifiers that are not constants are written with underscores like_this, whereas multiword constants are written LikeThis or LIKE_THIS.

2.4.1 Case sensitivity

Ruby is a case-sensitive language. Lowercase letters and uppercase letters are distinct. They keyword end, for example, is completely different from the keyword END.

2.4.2 Unicode characters in Identifiers

Ruby's rules for forming identifiers are defined in terms of ASCII characters that are not allowed. In general, all characters outside of the ASCII characters that appear to be punctuation.

2.4.3 Punctuation in identifiers

A number of Ruby's operators are implemented as methods, so that classes can redefine them for their own purposes. It is therefore possible to use certain operators as methods names as well. In this context, the punctuation character or characters of the operator are treated as identifiers rather than operators.

2.5 Whitespace

Spaces, tabs, and newlines are not tokens themselves but are used to separate tokens that would otherwise merge into a single token. Aside from this basic token-separating function, most whitespace is ignored by the Ruby interpreter and is simply used to format programs so that they are easy to read and understand. Not all whitespace is ignored, however. Some is required, and some whitespace is actually forbidden. Ruby's grammar is expressive but complex.

2.5.1 Newlines as statement terminators

The most common form of whitespace dependency has to do with the newlines as statement terminators. In languages like C and Java, every statement must be terminated with a semicolon. You can use semicolons to terminate statements in Ruby too, but this is only required if you put more than one statement on the same line. Convention dictates that semicolons be omitted elsewhere.

Without explicit semicolons, the Ruby interpreter must figure out on its own where statements ends. If the Ruby code on a line is a syntactically complete statement, Ruby uses the newline as the statement terminator. If the statement is not complete, then Ruby continues parsing the statement on the next line.

This is no problem if all your statements fit on a single line. When they don't, however, you must take care that you break the line in such a way that the Ruby interpreter cannot interpret the first line as a statement of its own. This is where the whitespace dependency lies: your program may behave differently depending on where you insert a newline.

2.5.2 Spaces and method invocations

Ruby's grammar allows the parentheses around method invocations to be omitted in certain circumstances. This allows Ruby methods to be used as if they were statements, which is an important part of Ruby's elegance. Unfortunately, however, it opens up a pernicious whitespace dependency. Consider the following two lines, which differ only by a single space. The first lines passes the value 5 to the function f and then adds 1 to the result. Since the second line has a space after the function name, Ruby assumes that the parentheses around the method call have been omitted. The parentheses that appear after the space are used to group a subexpression, but the entire expression (3 + 2) + 1 is used as the method argument. If warnings are enabled (with -w), Ruby issues

a warning whenever it sees ambiguous code like this.

The solution to this whitespace dependency is straightforward:

- Never put a space between a method name and the opening parenthesis.
- If the first argument to a method begins with an open parenthesis, always use parentheses in the method invocation.
- Always run the Ruby interpreter with the -w option so it will warn you if you forget either of these above.

3 Syntactic Structure

So far, we've discussed the tokens of a Ruby program and the characters that make them up. Now we have to move on to briefly describe how those lexical tokens combine into the larger syntactic structures of a Ruby programs, from the simplest expressions to the largest modules.

The basic unit of syntax in Ruby is the *expression*. The Ruby interpreter *evaluates* expressions, producing values. The simplest expressions are *primary expressions*, which represent values directly. Number and string literals, described earlier in this chapter, are primary expressions. Other primary expressions include certain keywords such as true, false, nil, and self. Variable references are also primary expressions; they evaluate to the value of the variable.

Expressions can be combined with Ruby's keywords to create *statements*, such as the if statement for conditionally executing code and the while statement for repeatedly executing code. In Ruby, these statements are technically expressions, but there is still a useful distinction between expressions that affect the control flow of a program and those that do not.

In all but the most trivial programs, we usually need to group expressions and statements into parametrized units so that they can executed repeatdly and operate on varying inputs. You may know these parametrized units as functions, procedures, or subroutines. Since Ruby is an object-oriented language, they are called *methods*. Methods, along with related structures called *procs* and *lambda*.

Finally, groups of methods that are designed to interoperate can be combined into *classes*, and groups of related classes and methods that are independent of those classes can be organized into *modules*.

Ruby programs have a block structure. Module, class, and method definitions, and most of Ruby's statements, include bloccks of nested. These blocks are delimited by keywoprds or punctuation and, by convention, are indented two

spaces relative to the delimiters. There are two kinds of blocks in Ruby programs. These blocks are the chunks of code associated with or passed to iterator methods:

```
3.times {print "Ruby!"}
```

In this code, the curly braces and the code inside them are the block associated with the iterator method invocation 3.times. Formal blocks of this kind may be delimited with curly braces, or they may be delimited with the keywords do and end:

```
1.upto(10) do |x|
    print x
end
```

do and end delimiters are usually used when the block is written on more than one line. Note the two space indentation of the code within the block. Blocks are covered after.

To avoid ambiguity with these true blocks, we can call the other kind of block a *block*. A body is just the list of statements that comprise the body of a class definition, a method definition, a while loop, or whatever. Bodies are never delimited with curly braces in Ruby - keywords usually serve as the delimiters instead.

4 File Structure

There are only a few rules about a file of Ruby code must be structured. These rules are related to the deployment of Ruby programs and are directly relevant to the language itself. First, if a Ruby program contains a "shebang" comment, to tell the (Unix-like) operating system how to execute it, that comment must appear on the first line.

Second, if a Ruby program contains a "coding" comment, that comment must appear on the first line or on the second line if the first line is a shebang.

Third, if a file contains a line that consists of the single token __END__ with no whitespace before or after, then the Ruby interpreter stops processing the file at that point. The remainder of the file may contain arbitrary data that the program can read using the IO stream object DATA.

Ruby programs are not required to fit in a single file. Many programs load additional Ruby code from external libraries, for example. Programs use require to load code from another file require searches for specified modules of

code against a search path, and prevents any given module from being loaded more than once.

5 Program Encoding

At the lowest level, a Ruby program is simply a sequence of characters. Ruby's lexical rules are defined using characters of the ASCIII character set. Comments begin with the `#` character (ASCII code 35), for example, and allowed whitespace characters are horizontal tab (ASCII 9), newline (10), vertical tab(11), form feed(12), carriage return (13), and space (32). All Ruby keywords are written using ASCII characters.

6 Program Execution

Ruby is a scripting language. This means that Ruby programs are simply lists, or scripts, of statements to be executed. By default, these statements are executed sequentially, in the order they appear. Ruby's control structures alter this default execution order and allow statement to be executed conditionally or repeatedly, for example.

Programmers who are used to traditionally static compiled languages like C or Java find this slightly confusing. There is no special main method in Ruby from which execution begins. The Ruby interpreter is given a script of statements to execute, and it begins executing at the first line and continues to the last line.