1.2. Basic Ruby Syntax and Semantics

In the previous pages, we have already seen that Ruby is a *pure, dynamic, OOP* language.

Let's look briefly at some other attributes before summarizing the syntax and semantics.

Ruby is an *agile* language. It is "malleable" and encourages frequent, easy refactoring.

Ruby is an *interpreted* language. Of course, there may be later implementations of a Ruby compiler for performance reasons, but we maintain that an interpreter yields great benefits not only in rapid prototyping but also in the shortening of the development cycle overall.

Ruby is an *expression-oriented* language. Why use a statement when an expression will do? This means, for instance, that code becomes more compact as the common parts are factored out and repetition is removed.

Ruby is a *very high-level language* (*VHLL*). One principle behind the language design is that the computer should work for the programmer rather than vice versa. The "density" of Ruby means that sophisticated and complex operations can be carried out with relative ease as compared to lower-level languages.

Let's start by examining the overall look and feel of the language and some of its terminology. We'll briefly examine the nature of a Ruby program before looking at examples.

To begin with, Ruby is essentially a line-oriented language—more so than languages such as C but not so much as antique languages such as FORTRAN. Tokens can be crowded onto a single line as long as they are separated by whitespace as needed. Statements may occur more than one to a line if they are separated by semicolons; this is the only time the terminating semicolon is really needed. A line may be continued to the next line by ending with a backslash or by letting the parser know that the statement is not complete—for example, by ending a line with a comma.

There is no main program as such; execution proceeds in general from top to bottom. In more complex programs, there may be numerous definitions at the top followed by the (conceptual) main program at the bottom; but even in that case, execution proceeds from the top down because definitions in Ruby are executed.

1.2.1. Keywords and Identifiers

The keywords (or reserved words) in Ruby typically cannot be used for other purposes. These are as follows:

`BEGIN`	`END`	`alias`	`and`	`begin`
`break`	`case`	`class`	`def`	`defined?`
`do`	`else`	`elsif`	`end`	`ensure`
`false`	`for`	`if`	`in`	`module`
`next`	`nil`	`not`	`or`	`redo`
`rescue`	`retry`	`return`	`self`	`super`
`then`	`true`	`undef`	`unless`	`until`
`when`	`while`	`yield`		

Variables and other identifiers normally start with an alphabetic letter or a special modifier. The basic rules are as follows:

- Local variables (and pseudovariables such as `self` and `nil`) begin with a lowercase letter or an underscore.
- Global variables begin with a `$` (dollar sign).

- Instance variables (within an object) begin with an `@` (at sign).
- Class variables (within a class) begin with two `@` signs.
- Constants begin with capital letters.
- For purposes of forming identifiers, the underscore (`_`) may be used as a lowercase letter.
- Special variables starting with a dollar sign (such as `$1` and `$/`) are not dealt with here.

Here are some examples of each of these:

- Local variables— `alpha`, `_ident`, `some_var`
- Pseudovariables— `self`, `nil`, `__FILE__`
- Constants— `K6chip`, `Length`, `LENGTH`
- Instance variables— `@foobar`, `@thx1138`, `@NOT_CONST`
- Class variable— `@@phydeaux`, `@@my_var`, `@@NOT_CONST`
- Global variables— `$beta`, `$B12vitamin`, `$NOT_CONST`

1.2.2. Comments and Embedded Documentation

Comments in Ruby begin with a pound sign (#) outside a string or character constant and proceed to the end of the line:

```
x = y + 5 # This is a comment.
# This is another comment.
print "# But this isn't."
```

Embedded documentation is intended to be retrieved from the program text by an external tool. From the point of view of the interpreter, it is like a comment and can be used as such. Given two lines starting with `=begin` and `=end`, everything between those lines (inclusive) is ignored by the interpreter. (These can't be preceded by whitespace.)

```
=begin
The purpose of this program
is to cure cancer
and instigate world peace.
=end
```

1.2.3. Constants, Variables, and Types

In Ruby, variables do not have types, but the objects they refer to still have types. The simplest data types are character, numeric, and string.

Numeric constants are mostly intuitive, as are strings. Generally, a double-quoted string is subject to additional interpretation, and a single-quoted string is more "as is," allowing only an escaped backslash.

In double-quoted strings, we can do "interpolation" of variables and expressions as shown here:

```
a = 3
b = 79
puts "#{a} times #{b} = #{a*b}"    #   3 times 79 = 237
```

For more information on literals (numbers, strings, regular expressions, and so on), refer to later chapters.

There is a special kind of string worth mentioning, primarily useful in small scripts used to glue together larger programs. The command output string is sent to the operating system as a command to be executed, whereupon the output of the command is substituted back into the string. The simple form of this string uses the *grave accent* (sometimes called a *back-tick* or *back-quote*) as a beginning and ending delimiter; the more complex form uses the `%x` notation:

```
`whoami`
`ls -l`
%x[grep -i meta *.html | wc -l]
```

Regular expressions in Ruby look similar to character strings, but they are used differently. The usual delimiter is a slash character.

For those familiar with Perl, regular expression handling is similar in Ruby. Incidentally, we'll use the abbreviation *regex* throughout the remainder of the book; many people abbreviate it as *regexp*, but that is not as pronounceable. For details on regular expressions, see Chapter 3, "Working with Regular Expressions."

Arrays in Ruby are a powerful construct; they may contain data of any type or may even mix types. As we shall see in Chapter 8 ("Arrays, Hashes, and Other Enumerables"), all arrays are instances of the class `Array` and thus have a rich set of methods that can operate on them. An array constant is delimited by brackets; the following are all valid array expressions:

```
[1, 2, 3]
[1, 2, "buckle my shoe"]
[1, 2, [3,4], 5]
["alpha", "beta", "gamma", "delta"]
```

The second example shows an array containing both integers and strings; the third example in the preceding code shows a nested array, and the fourth example shows an array of strings. As in most languages, arrays are zero-based; for instance, in the last array in the preceding code, `"gamma"` is element number 2. Arrays are dynamic and do not need to have a size specified when they are created.

Since the array of strings is so common (and so inconvenient to type), a special syntax has been set aside for it, similar to what we have seen before:

```
%w[alpha beta gamma delta]
%w(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)
%w/am is are was were be being been/
```

In these, the quotes and commas are not needed; only whitespace separates the individual elements. In the case of an element that contained whitespace, of course, this would not work.

An array variable can use brackets to index into the array. The resulting expression can be both examined and assigned to:

```
val = myarray[0]
print stats[j]
x[i] = x[i+1]
```

Another powerful construct in Ruby is the *hash*, also known in other circles as an *associative array* or *dictionary*. A hash is a set of associations between paired pieces of data; it is typically used as a lookup table or a kind of generalized array in which the index need not be an integer. Each hash is an instance of the class `Hash`.

A hash constant is typically represented between delimiting braces, with the symbol `=>` separating the individual keys and values. The key can be thought of as an index where the corresponding value is stored. There is no restriction on types of the keys or the corresponding values. Here are some hashes:

```
{1=>1, 2=>4, 3=>9, 4=>16, 5=>25, 6=>36}
{"cat"=>"cats", "ox"=>"oxen", "bacterium"=>"bacteria"}
{"hydrogen"=>1, "helium"=>2, "carbon"=>12}
{"odds"=>[1,3,5,7], "evens"=>[2,4,6,8]}
{"foo"=>123, [4,5,6]=>"my array", "867-5309"=>"Jenny"}
```

A hash variable can have its contents accessed by essentially the same bracket notation that arrays use:

```
print phone_numbers["Jenny"]
plurals["octopus"] = "octopi"
```

It should be stressed, however, that both arrays and hashes have many methods associated with them; these methods give them their real usefulness. The section "OOP in Ruby" later in the chapter will expand on this a little more.

1.2.4. Operators and Precedence

Now that we have established our most common data types, let's look at Ruby's operators. They are arranged here in order from highest to lowest precedence:

`::`	Scope
`[]`	Indexing
`**`	Exponentiation
`+ - ! ~`	Unary pos/neg, not, ...
`* / %`	Multiplication, division, ...
`+ -`	Addition/subtraction
`<< >>`	Logical shifts, ...
`&`	Bitwise and
`| ^`	Bitwise or, xor
`> >= < <=`	Comparison
`== === <=> != =~ !~`	Equality, inequality, ...
`&&`	Boolean and
`||`	Boolean or
`.. ...`	Range operators
`=` (also `+=, -=, ...`)	Assignment
`?:`	Ternary decision
`not`	Boolean negation
`and or`	Boolean and, or

Some of the preceding symbols serve more than one purpose; for example, the operator `<<` is a bitwise left shift but is also an append operator (for arrays, strings, and so on) and a marker for a here-document. Likewise the `+` is for numeric addition as well as for string concatenation. As we shall see later, many of these operators are just shortcuts for method names.

Now we have defined most of our data types and many of the possible operations on them. Before going any further, let's look at a sample program.

1.2.5. A Sample Program

In a tutorial, the first program is always `Hello, world!` But in a whirlwind tour like this one, let's start with something slightly more advanced. Here is a small interactive console-based program to convert between Fahrenheit and Celsius temperatures:

```
print "Please enter a temperature and scale (C or F): "
str = gets
exit if str.nil? or str.empty?
str.chomp!
temp, scale = str.split(" ")

abort "#{temp} is not a valid number." if temp !~ /-?\d+/

temp = temp.to_f
case scale
  when "C", "c"
    f = 1.8*temp + 32
  when "F", "f"
    c = (5.0/9.0)*(temp-32)
else
  abort "Must specify C or F."
end

if f.nil?
  print "#{c} degrees C\n"
else
  print "#{f} degrees F\n"
end
```

Here are some examples of running this program. These show that the program can convert from Fahrenheit to Celsius, convert from Celsius to Fahrenheit, and handle an invalid scale or an invalid number:

```
Please enter a temperature and scale (C or F): 98.6 F
37.0 degrees C

Please enter a temperature and scale (C or F): 100 C
212.0 degrees F

Please enter a temperature and scale (C or F): 92 G
Must specify C or F.

Please enter a temperature and scale (C or F): junk F
junk is not a valid number.
```

Now, as for the mechanics of the program: We begin with a `print` statement, which is actually a call to the `Kernel` method `print`, to write to standard output. This is an easy way of leaving the cursor "hanging" at the end of the line.

Following this, we call `gets` (get string from standard input), assigning the value to `str`. We then do a `chomp!` to remove the trailing newline.

Note that any apparently "free-standing" function calls such as `print` and `gets` are actually methods of `Object` (probably originating in `Kernel`). In the same way, `chop` is a method called with `str` as a receiver. Method calls in Ruby usually can omit the parentheses; `print "foo"` is the same as `print("foo")`.

The variable `str` holds a character string, but there is no reason it could not hold some other type instead. In Ruby, data have types, but variables do not. A variable springs into existence as soon as the interpreter sees an assignment to that variable; there are no "variable declarations" as such.

The `exit` is a call to a method that terminates the program. On this same line there is a control structure called an *if-modifier*. This is like the `if` statement that exists in most languages, but backwards; it comes after the action, does not permit an `else`, and does not require closing. As for the condition, we are checking two things: Does `str` have a value (is it non-`nil`), and is it a non-null string? In the case of an immediate end-of-file, our first condition will hold; in the case of a newline with no preceding data, the second condition will hold.

The same statement could be written this way:

```
exit if not str or not str[0]
```

The reason these tests work is that a variable can have a `nil` value, and `nil` evaluates to false in Ruby. In fact, `nil` and `false` evaluate as false, and everything else evaluates as true. Specifically, the null string `""` and the number 0 do *not* evaluate as false.

The next statement performs a `chomp!` operation on the string (to remove the trailing newline). The exclamation point as a prefix serves as a warning that the operation actually changes the value of its receiver rather than just returning a value. The exclamation point is used in many such instances to remind the programmer that a method has a side effect or is more "dangerous" than its unmarked counterpart. The method `chomp`, for example, returns the same result but does not modify its receiver.

The next statement is an example of multiple assignment. The `split` method splits the string into an array of values, using the space as a delimiter. The two assignable entities on the left-hand side will be assigned the respective values resulting on the right-hand side.

The `if` statement that follows uses a simple regex to determine whether the number is valid; if the string fails to match a pattern consisting of an optional minus sign followed by one or more digits, it is an invalid number (for our purposes), and the program exits. Note that the `if` statement is terminated by the keyword `end`; though it was not needed here, we could have had an `else` clause before the `end`. The keyword `then` is optional; we tend not to use it in this book.

The `to_f` method is used to convert the string to a floating point number. We are actually assigning this floating point value back to `temp`, which originally held a string.

The `case` statement chooses between three alternatives—the cases in which the user specified a `C`, specified an `F`, or used an invalid scale. In the first two instances, a calculation is done; in the third, we print an error and exit.

Ruby's `case` statement, by the way, is far more general than the example shown here. There is no limitation on the data types, and the expressions used are all arbitrary and may even be ranges or regular expressions.

There is nothing mysterious about the computation. But consider the fact that the variables `c` and `f` are referenced first inside the branches of the case. There are no declarations as such in Ruby; since a variable only comes into existence when it is assigned, this means that when we fall through the `case` statement, only one of these variables actually has a valid value.

We use this fact to determine after the fact which branch was followed, so that we can do a slightly different output in each instance. Testing `f` for a `nil` is effectively a test of whether the variable has a meaningful value. We do this here only to show that it can be done; obviously two different print statements could be used inside the case statement if we wanted.

The perceptive reader will notice that we used only "local" variables here. This might be confusing since their scope certainly appears to cover the entire program. What is happening here is that the variables are all local to the *top level* of the program (written *toplevel* by some). The variables appear global because there are no lower-level contexts in a program this simple; but if we declared classes and methods, these top-level variables would not be accessible within those.

1.2.6. Looping and Branching

Let's spend some time looking at control structures. We have already seen the simple `if` statement and the `if`-modifier; there are also corresponding structures based on the keyword `unless` (which also has an optional `else`), as well as expression-oriented forms of `if` and `unless`. We summarize all of these in Table 1.1.

Table 1.1. Conditional Statements

"if" Form	"unless" Form

`if x < 5 then`
`  statement1`
`end`
	`unless x >= 5 then`
	`  statement1`
	`end`

`if x < 5 then`
`  statement1`
`else`
`  statement2`
`end`
	`unless x < 5 then`
	`  statement2`
	`else`
	`  statement1`
	`end`

`statement1 if y == 3`	`statement1 unless y != 3`
`x = if a>0 then b else c end`	`x = unless a<=0 then c else b end`

In Table 1.1, the `if` and `unless` forms that are on the same table row have exactly the same function. Note that the keyword `then` may always be omitted except in the final (expression-oriented) cases. Note also that the modifier forms (in the third row) cannot have an `else` clause.

The `case` statement in Ruby is more powerful than in most languages. This multiway branch can even test for conditions other than equality—for example, a matched pattern. The test done by the case statement corresponds to the *case equality operator* (`===`), which has a behavior that varies from one object to another. Let's look at this example:

```
case "This is a character string."
  when "some value"
    puts "Branch 1"
  when "some other value"
    puts "Branch 2"
  when /char/
    puts "Branch 3"
  else
    puts "Branch 4"
end
```

The preceding code prints `Branch 3`. Why? It first tries to check for equality between the tested expression and one of the strings `"some value"` or `"some other value"`; this fails, so it proceeds. The third test is for the presence of a pattern within the tested expression; that pattern is there, so the test succeeds, and the third `print` statement is performed. The `else` clause always handles the default case in which none of the preceding tests succeeds.

If the tested expression is an integer, the compared value can be an integer range (for example, `3..8`). In this case, the expression is tested for membership in that range. In all instances, the first successful branch will be taken.

As for looping mechanisms, Ruby has a rich set. The `while` and `until` control structures are both pretest loops, and both work as expected: One specifies a continuation condition for the loop, and the other specifies a termination condition. They also occur in "modifier" form such as `if` and `unless`. There is also the `loop` method of the `Kernel` module (by default an infinite loop), and there are iterators associated with various classes.

The examples in Table 1.2 assume an array called `list`, defined something like this: `list = %w[alpha bravo charlie delta echo]`; they all step through the array and write out each element.

Table 1.2. Looping Constructs

```
# Loop 1 (while)
i=0
while i < list.size do
  print "#{list[i]} "
  i += 1
end
```

```
# Loop 2 (until)
i=0
until i == list.size do
  print "#{list[i]} "
  i += 1
end
```

```
# Loop 3 (for)
for x in list do
  print "#{x} "
end
```

```
# Loop 4 ('each' iterator)
list.each do |x|
  print "#{x} "
end
```

```
# Loop 5 ('loop' method)
i=0
n=list.size-1
loop do
  print "#{list[i]} "
  i += 1
  break if i > n
end
```

```
# Loop 6 ('loop' method)
i=0
n=list.size-1
loop do
  print "#{list[i]} "
  i += 1
  break unless i <= n
end
```

```
# Loop 7 ('times' iterator)
n=list.size
n.times do |i|
  print "#{list[i]} "
end
```

```
# Loop 8 ('upto' iterator)
n=list.size-1
0.upto(n) do |i|
  print "#{list[i]} "
end
```

```
# Loop 9 (for)
n=list.size-1
for
  i in 0..n do
  print "#{list[i]} "
end
```

```
# Loop 10 ('each_index')
list.each_index do |xt
  print "#{list[x]} "
end
```

Let's examine these in detail. Loops 1 and 2 are the "standard" forms of the `while` and `until` loops; they behave essentially the same, but their conditions are negations of each other. Loops 3 and 4 are the same thing in "post-test" versions; the test is performed at the end of the loop rather than at the beginning. Note that the use of `begin` and `end` in this context is strictly a kludge or hack; what is really happening is that a begin/end block (used for exception handling) is followed by a `while` or `until` modifier. For someone really wanting a post-test loop, however, this is effectively the same.

Loops 3 and 4 are arguably the "proper" way to write this loop. Note the simplicity of these two compared with the others; there is no explicit initialization and no explicit test or increment. This is because an array "knows" its own size, and the standard iterator `each` (loop 6) handles such details automatically. Indeed, loop 3 is merely an indirect reference to this same iterator because the `for` loop works for any object having the iterator `each` defined. The `for` loop is only a shorthand for a call to `each`; such a shorthand is frequently called "syntax sugar" because it offers a more convenient alternative to another syntactic form.

Loops 5 and 6 both use the `loop` construct; as we said previously, `loop` looks like a keyword introducing a control structure, but it is really a method of the module `Kernel`, not a control structure at all.

Loops 7 and 8 take advantage of the fact that the array has a numeric index; the `times` iterator executes a specified number of times, and the `upto` iterator carries its parameter up to the specified value. Neither of these is truly suitable for this instance.

Loop 9 is a `for` loop that operates specifically on the index values, using a range, and loop 10 likewise uses the `each_index` iterator to run through the list of array indices.

In the preceding examples, we have not laid enough emphasis on the "modifier" form of the `while` and `until` loops. These are frequently useful, and they have the virtue of being concise. The following are additional examples, both of which mean the same thing:

```
perform_task() until finished

perform_task() while not finished
```

Another fact is largely ignored in Table 1.2: Loops do not always run smoothly from beginning to end, in a predictable number of iterations, or ending in a single predictable way. We need ways to control these loops further.

The first way is the `break` keyword, shown in loops 5 and 6 above. This is used to "break out" of a loop; in the case of nested loops, only the innermost one is halted. This will be intuitive for C programmers.

The keyword `retry` is used in two situations—in the context of an iterator and in the context of a begin/end block (exception handling). Within the body of any iterator (or `for` loop), it forces the iterator to restart, re-evaluating any arguments passed to the iterator. Note that it does not work for loops in general (`while` and `until`).

The `redo` keyword is the generalized form of `retry` for loops. It works for `while` and `until` loops just as `retry` works for iterators.

The `next` keyword effectively jumps to the end of the innermost loop and resumes execution from that point. It works for any loop or iterator.

The iterator is an important concept in Ruby, as we have already seen. What we have not seen is that the language allows user-defined iterators in addition to the predefined ones.

The default iterator for any object is called `each`. This is significant partly because it allows the `for` loop to be used. But iterators may be given different names and used for varying purposes.

As a crude example, consider this multipurpose iterator, which mimics a post-test loop (like C's do-while or Pascal's repeat-until):

```
def repeat(condition)
  yield
  retry if not condition
end
```

In this example, the keyword `yield` is used to call the block that is specified when the iterator is called in this way:

```
j=0
repeat (j >= 10) do
  j+=1
  puts j
end
```

It is also possible to pass parameters via `yield`, which will be substituted into the block's parameter list (between vertical bars). As a somewhat contrived example, the following iterator does nothing but generate the integers from `1` to `10`, and the call of the iterator generates the first ten cubes:

```
def my_sequence
  for i in 1..10 do
    yield i
  end
end

my_sequence {|x| puts x**3 }
```

Note that `do` and `end` may be substituted for the braces that delimit a block. There are differences, but they are fairly subtle.

1.2.7. Exceptions

Like many other modern programming languages, Ruby supports *exceptions*.

Exceptions are a means of handling errors that has significant advantages over older methods. Return codes are avoidable, as is the "spaghetti logic" that results from checking them, and the code that detects the error can be distinguished from the code that knows how to handle the error (since these are often separate anyway).

The `raise` statement raises an exception. Note that `raise` is not a reserved word but a method of the module `Kernel`. (There is an alias named `fail`.)

```
raise                                          # Example 1
raise "Some error message"                     # Example 2
raise ArgumentError                            # Example 3
raise ArgumentError, "Bad data"                # Example 4
raise ArgumentError.new("Bad data")            # Example 5
raise ArgumentError, "Bad data", caller[0] # Example 6
```

In the first example in the preceding code, the last exception encountered is re-raised. In example 2, a `RuntimeError` (the default error) is created using the message `Some error message`.

In example 3, an `ArgumentError` is raised; in example 4, this same error is raised with the message `Bad data` associated with it. Example 5 behaves exactly the same as example 4. Finally, example 6 adds traceback information of the form `"filename:line"` or `"filename:line:in `method'"` (as stored in the `caller` array).

Now, how do we handle exceptions in Ruby? The `begin-end` block is used for this purpose. The simplest form is a `begin-end` block with nothing but our code inside:

```
begin
  # No real purpose.
  # ...
end
```

This is of no value in catching errors. The block, however, may have one or more `rescue` clauses in it. If an error occurs at any point in the code, between `begin` and `rescue`, control will be passed immediately to the appropriate `rescue` clause.

```
begin
  x = Math.sqrt(y/z)
  # ...
rescue ArgumentError
  puts "Error taking square root."
rescue ZeroDivisionError
  puts "Attempted division by zero."
end
```

Essentially the same thing can be accomplished by this fragment:

```
begin
  x = Math.sqrt(y/z)
  # ...
rescue => err
  puts err
end
```

Here the variable `err` is used to store the value of the exception; printing it causes it to be translated to some meaningful character string. Note that since the error type is not specified, the `rescue` clause will catch any descendant of `StandardError`. The notation `rescue => variable` can be used with or without an error type before the `=>` symbol.

In the event that error types are specified, it may be that an exception does not match any of these types. For that situation, we are allowed to use an `else` clause after all the `rescue` clauses.

```
begin
```

```
  # Error-prone code...
rescue Type1
  # ...
rescue Type2
  # ...
else
  # Other exceptions...
end
```

In many cases, we want to do some kind of recovery. In that event, the keyword `retry` (within the body of a `rescue` clause) restarts the `begin` block and tries those operations again:

```
begin
  # Error-prone code...
rescue
  # Attempt recovery...
  retry # Now try again
end
```

Finally, it is sometimes necessary to write code that "cleans up" after a `begin-end` block. In the event this is necessary, an `ensure` clause can be specified:

```
begin
  # Error-prone code...
rescue
  # Handle exceptions
ensure
  # This code is always executed
end
```

The code in an `ensure` clause is always executed before the `begin-end` block exits. This happens regardless of whether an exception occurred.

Exceptions may be caught in two other ways. First, there is a modifier form of the `rescue` clause:

```
x = a/b rescue puts("Division by zero!")
```

In addition, the body of a method definition is an implicit `begin-end` block; the `begin` is omitted, and the entire body of the method is subject to exception handling, ending with the `end` of the method:

```
def some_method
  # Code...
rescue
  # Recovery...
end
```

This sums up the discussion of exception handling as well as the discussion of fundamental syntax and semantics.

There are numerous aspects of Ruby we have not discussed here. The rest of this chapter is devoted to the more advanced features of the language, including a collection of Ruby lore that will help the intermediate programmer learn to "think in Ruby."