Ruby User's Guide

About the guide

About the guide

This introductory user's guide is mirrored in various places and is available in several translations. The current English language version is kept on rubyist.net. If you come across an out-of-date version, please notify the webmaster where the mirror is hosted.

Document history

- Original Japanese version by matz.
- First English translation by GOTO Kentaro & Julian Fondren.
- Re-translation and added material by Mark Slagell.

Ruby User's Guide

Contents & Chapters

Ruby is "an interpreted scripting language for quick and easy object-oriented programming"

Ruby User's Guide

What is ruby?

Ruby is "an interpreted scripting language for quick and easy object-oriented programming" -- what does this mean?

interpreted scripting language:

- ability to make operating system calls directly
- powerful string operations and regular expressions
- immediate feedback during development

quick and easy:

- variable declarations are unnecessary
- variables are not typed
- syntax is simple and consistent
- memory management is automatic

object oriented programming:

- everything is an object
- classes, methods, inheritance, etc.
- singleton methods
- "mixin" functionality by module
- iterators and closures

also:

- multiple precision integers
- convenient exception processing
- dynamic loading
- threading support

If you are unfamiliar with some of the concepts above, read on, and don't worry. The mantra of the ruby language is *quick and easy*.

Ruby User's Guide

Getting started

First, you'll want to check whether ruby is installed. From the shell prompt (denoted here by "`%`", so don't type the `%`), type

```
% ruby -v
```

(`-v` tells the interpreter to print the version of ruby), then press the *Enter* key. If ruby is installed, you will see a message something like the following:

```
% ruby -v
ruby 1.8.3 (2005-09-21) [i586-linux]
```

If ruby is not installed, you can ask your administrator to install it, or you can do it yourself, since ruby is free software with no restrictions on its installation or use.

Now, let's play with ruby. You can place a ruby program directly on the command line using the `-e` option:

```
% ruby -e 'puts "hello world"'
hello world
```

More conventionally, a ruby program can be stored in a file.

```
% echo "puts 'hello world'" > hello.rb
% ruby hello.rb
hello world
```

When writing more substantial code than this, you will want to use a real text editor!

Some surprisingly complex and useful things can be done with miniature programs that fit in a command line. For example, this one replaces `foo` with `bar` in all C source and header files in the current working directory, backing up the original files with ".bak" appended:

```
% ruby -i.bak -pe 'sub "foo", "bar"' *.[ch]
```

This program works like the UNIX `cat` command (but works slower than `cat`):

```
% ruby -pe 0 file
```

Ruby User's Guide

Simple examples

Let's write a function to compute factorials. The mathematical definition of `n` factorial is:

```
n! = 1                (when n==0)
   = n * (n-1)!       (otherwise)
```

In ruby, this can be written as:

```
def fact(n)
  if n == 0
    1
  else
    n * fact(n-1)
  end
end
```

You may notice the repeated occurrence of `end`. Ruby has been called "Algol-like" because of this. (Actually, the syntax of ruby more closely mimics that of a langage named Eiffel.) You may also notice the lack of a `return` statement. It is unneeded because a ruby function returns the last thing that was evaluated in it. Use of a `return` statement here is permissible but unnecessary.

Let's try out our factorial function. Adding one line of code gives us a working program:

```
# Program to find the factorial of a number
# Save this as fact.rb

def fact(n)
  if n == 0
    1
  else
    n * fact(n-1)
  end
end

puts fact(ARGV[0].to_i)
```

Here, ARGV is an array which contains the command line arguments, and `to_i` converts a character string to an integer.

```
% ruby fact.rb 1
1
% ruby fact.rb 5
120
```

Does it work with an argument of 40? It would make your calculator overflow...

```
% ruby fact.rb 40
815915283247897734345611269596115894272000000000
```

It does work. Indeed, ruby can deal with any integer which is allowed by your machine's memory. So 400! can be calculated:

```
% ruby fact.rb 400
64034522846623895262347970319503005850702583026002959458684
44594280239716918683143627847864746326467629435057503585681
08482981628835174352289619886468029979373416541508381624264
61942352307046244325015114448670890662773914918117331955996
44070954967134529047702032243491121079759328079510154537266
72516278778900093497637657103263503315339653498683868313393
52024373788157786791506311858702618270169819740062983025308
59129834616227230455833952075961150530223608681043329725519
48526744322324386699484224042325998055516106359423769613992
31917134063858996537970147827206606320217379472010321356624
61380907794230459736069956759583609615871512991382228657857
95493616176544804532220078258184008484364155912294542753848
03558374518022675900061399560145595206127211192918105032491
00800000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000
```

We cannot check the correctness at a glance, but it must be right. :-)

The input/evaluation loop

When you invoke ruby with no arguments, it reads commands from standard input and executes them after the end of input:

```
% ruby
puts "hello world"
puts "good-bye world"
^D
hello world
good-bye world
```

The *^D* above means control-D, a conventional way to signal end-of-input in a Unix context. In DOS/Windows, try pressing *F6* or *^Z* instead.

Ruby also comes with a program called `eval.rb` that allows you to enter ruby code from the keyboard in an interactive loop, showing you the results as you go. It will be used extensively through the rest of this guide.

If you have an ANSI-compliant terminal (this is almost certainly true if you are running some flavor of UNIX; under old versions of DOS you need to have installed `ANSI.SYS` or `ANSI.COM`; Windows XP, unfortunately, has now made this nearly impossible), you should use this enhanced `eval.rb` that adds visual indenting assistance, warning reports, and color highlighting. Otherwise, look in the `sample` subdirectory of the ruby distribution for the non-ANSI version that works on any terminal. Here is a short `eval.rb` session:

```
% ruby eval.rb
ruby> puts "Hello, world."
Hello, world.
   nil
ruby> exit
```

`hello world` is produced by `puts`. The next line, in this case `nil`, reports on whatever was last evaluated; ruby does not distinguish between *statements* and *expressions*, so evaluating a piece of code basically means the same thing as executing it. Here, `nil` indicates that `puts` does not return a meaningful value. Note that we can leave this interpreter loop by saying `exit`, although `^D` still works too.

Throughout this guide, "`ruby>`" denotes the input prompt for our useful little `eval.rb` program.

Ruby User's Guide

Strings

Ruby deals with strings as well as numerical data. A string may be double-quoted ("...") or single-quoted ('...').

```
ruby> "abc"
   "abc"
ruby> 'abc'
   "abc"
```

Double- and single-quoting have different effects in some cases. A double-quoted string allows character escapes by a leading backslash, and the evaluation of embedded expressions using `#{}`. A single-quoted string does not do this interpreting; what you see is what you get. Examples:

```
ruby> puts "a\nb\nc"
a
b
c
   nil
ruby> puts 'a\nb\n'
a\nb\nc
   nil
ruby> "\n"
   "\n"
ruby> '\n'
   "\\n"
ruby> "\001"
   "\001"
ruby> '\001'
   "\\001"
ruby> "abcd #{5*3} efg"
   "abcd 15 efg"
ruby> var = " abc "
   " abc "
ruby> "1234#{var}5678"
   "1234 abc 5678"
```

Ruby's string handling is smarter and more intuitive than C's. For instance, you can concatenate strings with `+`, and repeat a string many times with `*`:

```
ruby> "foo" + "bar"
   "foobar"
ruby> "foo" * 2
   "foofoo"
```

Concatenating strings is much more awkward in C because of the need for explicit memory management:

```
char *s = malloc(strlen(s1)+strlen(s2)+1);
strcpy(s, s1);
strcat(s, s2);
/* ... */
```

```
free(s);
```

But using ruby, we do not have to consider the space occupied by a string. We are free from all memory management.

Here are some things you can do with strings.

Concatenation:

```
ruby> word = "fo" + "o"
   "foo"
```

Repetition:

```
ruby> word = word * 2
   "foofoo"
```

Extracting characters (note that characters are integers in ruby):

```
ruby> word[0]
   102            # 102 is ASCII code of `f'
ruby> word[-1]
   111            # 111 is ASCII code of `o'
```

(Negative indices mean offsets from the end of a string, rather than the beginning.)

Extracting substrings:

```
ruby> herb = "parsley"
   "parsley"
ruby> herb[0,1]
   "p"
ruby> herb[-2,2]
   "ey"
ruby> herb[0..3]
   "pars"
ruby> herb[-5..-2]
   "rsle"
```

Testing for equality:

```
ruby> "foo" == "foo"
   true
ruby> "foo" == "bar"
   false
```

Note: In ruby 1.0, results of the above are reported in uppercase, e.g. TRUE.

Now, let's put some of these features to use. This puzzle is "guess the word," but perhaps the word "puzzle" is too dignified for what is to follow ;-)

```
# save this as guess.rb
words = ['foobar', 'baz', 'quux']
secret = words[rand(3)]

print "guess? "
while guess = STDIN.gets
  guess.chop!
  if guess == secret
    puts "You win!"
    break
  else
    puts "Sorry, you lose."
  end
  print "guess? "
end
puts "The word was ", secret, "."
```

For now, don't worry too much about the details of this code. Here is what a run of the puzzle program looks like.

```
% ruby guess.rb
guess? foobar
Sorry, you lose.
guess? quux
Sorry, you lose.
guess? ^D
The word was baz.
```

(I should have done a bit better, considering the 1/3 probability of success.)

Ruby User's Guide

Regular expressions

Let's put together a more interesting program. This time we test whether a string fits a description, encoded into a concise *pattern*.

There are some characters and character combinations that have special meaning in these patterns, including:

`[]`	range specificication (e.g., `[a-z]` means a letter in the range `a` to `z`)
`\w`	letter or digit; same as `[0-9A-Za-z]`
`\W`	neither letter or digit
`\s`	space character; same as `[ \t\n\r\f]`
`\S`	non-space character
`\d`	digit character; same as `[0-9]`
`\D`	non-digit character
`\b`	backspace (0x08) (only if in a range specification)
`\b`	word boundary (if not in a range specification)
`\B`	non-word boundary
`*`	zero or more repetitions of the preceding
`+`	one or more repetitions of the preceding
`{m,n}`	at least m and at most n repetitions of the preceding
`?`	at most one repetition of the preceding; same as `{0,1}`
`|`	either preceding or next expression may match
`()`	grouping

The common term for patterns that use this strange vocabulary is *regular expressions*. In ruby, as in Perl, they are generally surrounded by forward slashes rather than double quotes. If you have never worked with regular expressions before, they probably look anything but *regular*, but you would be wise to spend some time getting familiar with them. They have an efficient expressive power that will save you headaches (and many lines of code) whenever you need to do pattern matching, searching, or other manipulations on text strings.

For example, suppose we want to test whether a string fits this description: "Starts with lower case f, which is immediately followed by exactly one upper case letter, and optionally more junk after that, as long as there are no more lower case characters." If you're an experienced C programmer, you've probably already written about a dozen lines of code in your head, right? Admit it; you can hardly help yourself. But in ruby you need only request that your string be tested against the regular expression `/^f[A-Z][^a-z]*$/`.

How about "Contains a hexadecimal number enclosed in angle brackets"? No problem.

```
ruby> def chab(s)   # "contains hex in angle brackets"
    |    (s =~ /<0(x|X)(\d|[a-f]|[A-F])+>/) != nil
```

```
      | end
  nil
ruby> chab "Not this one."
  false
ruby> chab "Maybe this? {0x35}"    # wrong kind of brackets
  false
ruby> chab "Or this? <0x38z7e>"    # bogus hex digit
  false
ruby> chab "Okay, this: <0xfc0004>."
  true
```

Though regular expressions can be puzzling at first glance, you will quickly gain satisfaction in being able to express yourself so economically.

Here is a little program to help you experiment with regular expressions. Store it as `regx.rb` and run it by typing `"ruby regx.rb"` at the command line.

```
# Requires an ANSI terminal!

st = "\033[7m"
en = "\033[m"

puts "Enter an empty string at any time to exit."

while true
  print "str> "; STDOUT.flush; str = gets.chop
  break if str.empty?
  print "pat> "; STDOUT.flush; pat = gets.chop
  break if pat.empty?
  re = Regexp.new(pat)
  puts str.gsub(re,"#{st}\\&#{en}")
end
```

The program requires input twice, once for a string and once for a regular expression. The string is tested against the regular expression, then displayed with all the matching parts highlighted in reverse video. Don't mind details now; an analysis of this code will come soon.

```
str> foobar
pat> ^fo+
foobar
~~~
```

What you see above as red text will appear as reverse video in the program output. The "~~~" lines are for the benefit of those using text-based browsers.

Let's try several more inputs.

```
str> abc012dbcd555
pat> \d
abc012dbcd555
   ~~~    ~~~
```

If that surprised you, refer to the table at the top of this page: `\d` has no relationship to the character `d`, but rather matches a single digit.

What if there is more than one way to correctly match the pattern?

```
str> foozboozer
pat> f.*z
foozboozer
~~~~~~~~
```

`foozbooz` is matched instead of just `fooz`, since a regular expression maches the longest possible substring.

Here is a pattern to isolate a colon-delimited time field.

```
str> Wed Feb  7 08:58:04 JST 1996
pat> [0-9]+:[0-9]+(:[0-9]+)?
Wed Feb  7 08:58:04 JST 1996
           ~~~~~~~~
```

"`=~`" is a matching operator with respect to regular expressions; it returns the position in a string where a match was found, or `nil` if the pattern did not match.

```
ruby> "abcdef" =~ /d/
   3
ruby> "aaaaaa" =~ /d/
   nil
```

Ruby User's Guide

Arrays

You can create an *array* by listing some items within square brackets (`[]`) and separating them with commas. Ruby's arrays can accomodate diverse object types.

```
ruby> ary = [1, 2, "3"]
   [1, 2, "3"]
```

Arrays can be concatenated or repeated just as strings can.

```
ruby> ary + ["foo", "bar"]
   [1, 2, "3", "foo", "bar"]
ruby> ary * 2
   [1, 2, "3", 1, 2, "3"]
```

We can use index numbers to refer to any part of a array.

```
ruby> ary[0]
   1
ruby> ary[0,2]
   [1, 2]
ruby> ary[0..1]
   [1, 2]
ruby> ary[-2]
   2
ruby> ary[-2,2]
   [2, "3"]
ruby> ary[-2..-1]
   [2, "3"]
```

(Negative indices mean offsets from the end of an array, rather than the beginning.)

Arrays can be converted to and from strings, using `join` and `split` respecitvely:

```
ruby> str = ary.join(":")
   "1:2:3"
ruby> str.split(":")
   ["1", "2", "3"]
```

Hashes

An associative array has elements that are accessed not by sequential index numbers, but by *keys* which can have any sort of value. Such an array is sometimes called a *hash* or *dictionary*; in the ruby world, we prefer the term *hash*. A hash can be constructed by quoting pairs of items within curly braces (`{}`). You use a key to find something in a hash, much as you use an index to find something in an array.

```
ruby> h = {1 => 2, "2" => "4"}
   {1=>2, "2"=>"4"}
ruby> h[1]
```

```
   2
ruby> h["2"]
   "4"
ruby> h[5]
   nil
ruby> h[5] = 10    # appending an entry
   10
ruby> h
   {5=>10, 1=>2, "2"=>"4"}
ruby> h.delete 1   # deleting an entry by key
   2
ruby> h[1]
   nil
ruby> h
   {5=>10, "2"=>"4"}
```

Ruby User's Guide

Back to the simple examples

Now let's take apart the code of some of our previous example programs.

The following appeared in the simple examples chapter.

```
def fact(n)
  if n == 0
    1
  else
    n * fact(n-1)
  end
end
puts fact(ARGV[0].to_i)
```

Because this is the first explanation, we examine each line individually.

Factorials

```
def fact(n)
```

In the first line, `def` is a statement to define a function (or, more precisely, a *method*; we'll talk more about what a method is in a later chapter). Here, it specifies that the function `fact` takes a single argument, referred to as `n`.

```
if n == 0
```

The `if` is for checking a condition. When the condition holds, the next bit of code is evaluated; otherwise whatever follows the `else` is evaluated.

```
1
```

The value of `if` is 1 if the condition holds.

```
else
```

If the condition does not hold, the code from here to `end` is evaluated.

```
n * fact(n-1)
```

If the condition is not satisfied, the value of `if` is the result of `n` times `fact(n-1)`.

```
end
```

The first `end` closes the `if` statement.

```
end
```

The second `end` closes the `def` statement.

```
puts fact(ARGV[0].to_i)
```

This invokes our `fact()` function using a value specified from the command line, and prints the result.

`ARGV` is an array which contains command line arguments. The members of `ARGV` are strings, so we must convert this into a integral number by `to_i`. Ruby does not convert strings into integers automatically like perl does.

What would happen if we fed this program a negative number? Do you see the problem? Can you fix it?

Strings

Next we examine the puzzle program from the chapter on strings. As this is somewhat longer, we number the lines for reference.

```
01 words = ['foobar', 'baz', 'quux']
02 secret = words[rand(3)]
03
04 print "guess? "
05 while guess = STDIN.gets
06   guess.chop!
07   if guess == secret
08     puts "You win!"
09     break
10   else
11     puts "Sorry, you lose."
12   end
13   print "guess? "
14 end
15 puts "the word is ", secret, "."
```

In this program, a new control structure, `while`, is used. The code between `while` and its corresponding `end` will execute repeatedly as long as some specified condition remains true. In this case, `guess=STDIN.gets` is both an active statement (collecting a line of user input and storing it as `guess`), and a condition (if there is no input, `guess`, which repesents the value of the whole `guess=STDIN.gets` expression, has a nil value, causing `while` to stop looping).

`STDIN` is the standard input object. Usually, `guess=gets` does the same thing as `guess=STDIN.gets`.

`rand(3)` in line 2 returns a random number in the range 0 to 2. This random number is used to extract one of the members of the array `words`.

In line 5 we read one line from standard input by the method `STDIN.gets`. If *EOF* (end of file) occurs while getting the line, `gets` returns `nil`. So the code associated with this `while` will repeat until it sees *^D* (try *^Z* or *F6* under DOS/Windows), signifying the end of input.

`guess.chop!` in line 6 deletes the last character from `guess`; in this case it will always be a *newline* character, gets includes that character to reflect the user's *Return* keystroke, but we're not interested in it.

In line 15 we print the secret word. We have written this as a puts (put string) statement with two arguments, which are printed one after the other; but it would have been equally effective to do it with a single argument, writing secret as #{secret} to make it clear that it is a variable to be evaluated, not a literal word to be printed:

```
puts "the word is #{secret}."
```

Many programmers feel this is a cleaner way to express output; it builds a single string and presents it as a single argument to puts.

Also, we are by now used to the idea of using puts for standard script output, but this script uses print instead, in lines 4 and 13. They are not quite the same thing. print outputs exactly what it is given; puts also ensures that the output line ends. Using print in lines 4 and 13 leaves the cursor next to what was just printed, rather than moving it to the beginning of the next line. This creates a recognizable prompt for user input. In general, the four output calls below are equivalent:

```
# newline is implicitly added by puts if there isn't one
already:
puts  "Darwin's wife, Esmerelda, died in a fit of penguins."

# newline must be explicitly given to the print command:
print "Darwin's wife, Esmerelda, died in a fit of
penguins.\n"

# you can concatenate output with +:
print 'Darwin's wife, Esmerelda, died in a fit of
penguins.'+"\n"

# or concatenate by supplying more than one string:
print 'Darwin's wife, Esmerelda, died in a fit of penguins.',
"\n"
```

One possible gotcha: sometimes a text window is programmed to buffer output for the sake of speed, collecting individual characters and displaying them only when it is given a newline character. So if the guessing game script misbehaves by not showing the prompt lines until after the user supplies a guess, buffering is the likely culprit. To make sure this doesn't happen, you can flush the output as soon as you have printed the prompt. It tells the standard output device (an object named STDOUT), "don't wait; display what you have in your buffer right now."

```
04 print "guess? "; STDOUT.flush
  ...
13 print "guess? "; STDOUT.flush
```

And in fact, we were more careful with this in the next script.

Regular expressions

Finally we examine this program from the chapter on regular expressions.

```
01 st = "\033[7m"
02 en = "\033[m"
03
04 puts "Enter an empty string at any time to exit."
05
06 while true
07   print "str> "; STDOUT.flush; str=gets.chop
08   break if str.empty?
09   print "pat> "; STDOUT.flush; pat=gets.chop
10   break if pat.empty?
11   re = Regexp.new(pat)
12   puts str.gsub(re, "#{st}\\&#{en}")
13 end
```

In line 6, the condition for `while` is hardwired to `true`, so it forms what looks like an infinite loop. However we put `break` statements in the 8th and 10th lines to escape the loop. These two `break`s are also an example of "`if` modifiers." An `if` modifier executes the statement on its left hand side if and only if the specified condition is satisfied. This construction is unusual in that it operates logically from right to left, but it is provided because for many people it mimics a similar pattern in natural speech. It also has the advantage of brevity, as it needs no `end` statement to tell the interpreter how much of the following code is supposed to be conditional. An `if` modifier is conventionally used in situations where a statement and condition are short enough to fit comfortably together on one script line.

Note the difference in the user interface compared to the string-guessing script. This one lets the user quit by hitting the *Return* key on an empty line. We testing for emptiness of the input string, not for its nonexistence.

In lines 7 and 9 we have a "non-destructive" chop; again, we're getting rid of the unwanted newline character we always get from `gets`. Add the explanation point, and we have a "destructive" chop. What's the difference? In ruby, we conventionally attach '`!`' or '`?`' to the end of certain method names. The exclamation point (`!`, sometimes pronounced aloud as "bang!") indicates something potentially destructive, that is to say, something that can change the value of what it touches. `chop!` affects a string directly, but `chop` gives you a chopped copy without damaging the original. Here is an illustration of the difference.

```
ruby> s1 = "forth"
  "forth"
ruby> s1.chop!       # This changes s1.
  "fort"
ruby> s2 = s1.chop   # This puts a changed copy in s2,
  "for"
ruby> s1             # ... without disturbing s1.
  "fort"
```

You'll also sometimes see `chomp` and `chomp!` used. These are more selective: the end of a string gets bit off *only if it happens to be a newline.* So for example, `"XYZ".chomp!` does nothing. If you need a trick to remember the difference, think of a person or animal tasting something before deciding to take a bite, as opposed to an axe chopping indiscriminately.

The other method naming convention appears in lines 8 and 10. A question mark (`?`, sometimes pronounced aloud as "huh?") indicates a "predicate" method, one that can return either `true` or `false`.

Line 11 creates a regular expression object out of the string supplied by the user. The real work is finally done in line 12, which uses `gsub` to globally `sub`stitute each match of that expression with itself, but surrounded by ansi markups; also the same line outputs the results.

We could have broken up line 12 into separate lines like this:

```
highlighted = str.gsub(re,"#{st}\\&#{en}")
puts highlighted
```

or in "destructive" style:

```
str.gsub!(re,"#{st}\\&#{en}")
puts str
```

Look again at the last part of line 12. `st` and `en` were defined in lines 1-2 as the ANSI sequences that make text color-inverted and normal, respectively. In line 12 they are enclosed in `#{}` to ensure that they are actually interpreted as such (and we do not see the variable *names* printed instead). Between these we see `\\&`. This is a little tricky. Since the replacement string is in double quotes, the pair of backslashes will be interpreted as a single backslash; what `gsub` actually sees will be `\&`, and that happens to be a special code that refers to whatever matched the pattern in the first place. So the new string, when displayed, looks just like the old one, except that the parts that matched the given pattern are highlighted in inverse video.

Ruby User's Guide

Control structures

This chapter explores more of ruby's control structures.

case

We use the case statement to test a sequence of conditions. This is superficially similar to switch in C and Java but is considerably more powerful, as we shall see.

```
ruby> i=8
ruby> case i
    | when 1, 2..5
    |   puts "1..5"
    | when 6..10
    |   puts "6..10"
    | end
6..10
   nil
```

2..5 is an expression which means the *range* between 2 and 5, inclusive. The following expression tests whether the value of i falls within that range:

```
(2..5) === i
```

case internally uses the relationship operator === to check for several conditions at a time. In keeping with ruby's object oriented nature, === is interpreted suitably for the object that appeared in the when condition. For example, the following code tests string equality in the first when, and regular expression matching in the second when.

```
ruby> case 'abcdef'
    | when 'aaa', 'bbb'
    |   puts "aaa or bbb"
    | when /def/
    |   puts "includes /def/"
    | end
includes /def/
   nil
```

while

Ruby provides convenient ways to construct loops, although you will find in the next chapter that learning how to use *iterators* will make it unnecessary to write explicit loops very often.

A while is a repeated if. We used it in our word-guessing puzzle and in the regular expression programs (see the previous chapter); there, it took the form while *condition* ... end surrounding a block of code to be repeated while *condition* was true. But while and if can as easily be applied to individual statements:

```
ruby> i = 0
   0
ruby> puts "It's zero." if i==0
```

```
It's zero.
   nil
ruby> puts "It's negative." if i<0
   nil
ruby> puts i+=1 while i<3
1
2
3
   nil
```

Sometimes you want to negate a test condition. An `unless` is a negated `if`, and an `until` is a negated `while`. We'll leave it up to you to experiment with these.

There are four ways to interrupt the progress of a loop from inside. First, `break` means, as in C, to escape from the loop entirely. Second, `next` skips to the beginning of the next iteration of the loop (corresponding to C's `continue`). Third, ruby has `redo`, which restarts the current iteration. The following is C code illustrating the meanings of `break`, `next`, and `redo`:

```
while (condition) {
label_redo:
   goto label_next;        /* ruby's "next" */
   goto label_break;       /* ruby's "break" */
   goto label_redo;        /* ruby's "redo" */
   ...
   ...
label_next:
}
label_break:
...
```

The fourth way to get out of a loop from the inside is `return`. An evaluation of `return` causes escape not only from a loop but from the method that contains the loop. If an argument is given, it will be returned from the method call, otherwise `nil` is returned.

for

C programmers will be wondering by now how to make a "for" loop. Ruby's `for` can serve the same purpose, but adds some flexibility. The loop below runs once for each element in a *collection* (array, hash, numeric sequence, etc.), but doesn't make the programmer think about indices:

```
for elt in collection
  # ... here, elt refers to an element of the collection
end
```

The collection can be a range of values (this is what most people mean when they talk about a for loop):

```
ruby> for num in (4..6)
    |    puts num
    | end
4
5
```

```
6
   4..6
```

In this example we step through some array elements:

```
ruby> for elt in [100,-9.6,"pickle"]
    |    puts "#{elt}\t(#{elt.class})"
    | end
100    (Fixnum)
-9.6   (Float)
pickle (String)
   [100, -9.6, "pickle"]
```

But we're getting ahead of ourselves. `for` is really another way of writing `each`, which, it so happens, is our first example of an iterator. The following two forms are equivalent:

```
#  If you're used to C or Java, you might prefer this.
for element in collection
  ...
end

#  A Smalltalk programmer might prefer this.
collection.each {|element|
  ...
}
```

Iterators can often be substituted for conventional loops, and once you get used to them, they are generally easier to deal with. So let's move on and learn more about them.

Ruby User's Guide

Iterators

Iterators are not an original concept with ruby. They are in common use in object-oriented languages. They are also used in Lisp, though there they are not called iterators. However the concepet of iterator is an unfamiliar one for many so it should be explained in more detail.

The verb *iterate* means to do the same thing many times, you know, so an *iterator* is something that does the same thing many times.

When we write code, we need loops in various situations. In C, we code them using `for` or `while`. For example,

```
char *str;
for (str = "abcdefg"; *str != '\0'; str++) {
  /* process a character here */
}
```

C's `for(...)` syntax provides an abstraction to help with the creation of a loop, but the test of `*str` against a null character requires the programmer to know details about the internal structure of a string. This makes C feel like a low-level language. Higher level languages are marked by their more flexible support for iteration. Consider the following `sh` shell script:

```
#!/bin/sh

for i in *.[ch]; do
  # ... here would be something to do for each file
done
```

All the C source and header files in the current directory are processed, and the command shell handles the details of picking up and substituting file names one by one. I think this is working at a higher level than C, don't you?

But there is more to consider: while it is fine for a language to provide iterators for built-in data types, it is a disappointment if we must go back to writing low level loops to iterate over our own data types. In OOP, users often define one data type after another, so this could be a serious problem.

So every OOP language includes some facilities for iteration. Some languages provide a special class for this purpose; ruby allows us to define iterators directly.

Ruby's `String` type has some useful iterators:

```
ruby> "abc".each_byte{|c| printf "<%c>", c}; print "\n"
<a><b><c>
   nil
```

`each_byte` is an iterator for each character in the string. Each character is substituted into the local variable `c`. This can be translated into something that looks a lot like C code ...

```
ruby> s="abc";i=0
   0
ruby> while i<s.length
```

```
     |    printf "<%c>", s[i]; i+=1
     | end; print "\n"
<a><b><c>
   nil
```

... however, the `each_byte` iterator is both conceptually simpler and more likely to continue to work even if the `String` class happens to be radically modified in the future. One benefit of iterators is that they tend to be robust in the face of such changes; indeed that is a characteristic of good code in general. (Yes, have patience, we're about to talk about what *classes* are, too.)

Another iterator of `String` is `each_line`.

```
ruby> "a\nb\nc\n".each_line{|l| print l}
a
b
c
   nil
```

The tasks that would take most of the programming effort in C (finding line delimiters, generating substrings, etc.) are easily tackled using iterators.

The `for` statement appearing in the previous chapter does iteration by way of an `each` iterator. `String`'s `each` works the same as `each_line`, so let's rewrite the above example with `for`:

```
ruby> for l in "a\nb\nc\n"
     |   print l
     | end
a
b
c
   nil
```

We can use a control structure `retry` in conjunction with an iterated loop, and it will retry the loop from the beginning.

```
ruby> c=0
   0
ruby> for i in 0..4
     |   print i
     |   if i == 2 and c == 0
     |     c = 1
     |     print "\n"
     |     retry
     |   end
     | end; print "\n"
012
01234
   nil
```

Replacing `retry` in the above example with `redo` causes just the current iteration of the loop to be redone, with this output:

```
012
234
```

`yield` occurs sometimes in a definition of an iterator. `yield` moves control to the block of code that is passed to the iterator (this will be explored in more detail in the chapter about procedure objects). The following example defines an iterator `repeat`, which repeats a block of code the number of times specified in an argument.

```
ruby> def repeat(num)
    |   while num > 0
    |     yield
    |     num -= 1
    |   end
    | end
   nil
ruby> repeat(3) { puts "foo" }
foo
foo
foo
   nil
```

With `retry`, one can define an iterator which works something like ruby's standard `while`.

```
ruby> def WHILE(cond)
    |   return if not cond
    |   yield
    |   retry
    | end
   nil
ruby> i=0; WHILE(i<3) { print i; i+=1 }
012   nil
```

Do you understand what an iterator is? There are a few restrictions, but you can write your original iterators; and in fact, whenever you define a new data type, it is often convenient to define suitable iterators to go with it. In this sense, the above examples are not terribly useful. We can talk about practical iterators after we have a better understanding of what classes are.

Ruby User's Guide

Object-oriented thinking

Object oriented is a catchy phrase. To call anything object oriented can make you sound pretty smart. Ruby claims to be an object oriented scripting language; but what exactly does "object oriented" mean?

There have been a variety of answers to that question, all of which probably boil down to about the same thing. Rather than sum it too quickly, let's think for a moment about the traditional programming paradigm.

Traditionally, a programming problem is attacked by coming up with some kinds of *data representations*, and *procedures* that operate on that data. Under this model, data is inert, passive, and helpless; it sits at the complete mercy of a large procedural body, which is active, logical, and all-powerful.

The problem with this approach is that programs are written by programmers, who are only human and can only keep so much detail clear in their heads at any one time. As a project gets larger, its procedural core grows to the point where it is difficult to remember how the whole thing works. Minor lapses of thinking and typographical errors become more likely to result in well-concealed bugs. Complex and unintended interactions begin to emerge within the procedural core, and maintaining it becomes like trying to carry around an angry squid without letting any tentacles touch your face. There are guidelines for programming that can help to minimize and localize bugs within this traditional paradigm, but there is a better solution that involves fundamentally changing the way we work.

What object-oriented programming does is to let us delegate most of the mundane and repetitive logical work *to the data itself*; it changes our concept of data from *passive* to *active*. Put another way,

- We stop treating each piece of data as a box with an open lid that lets us reach in and throw things around.
- We start treating each piece of data as a working machine with a closed lid and a few well-marked switches and dials.

What is described above as a "machine" may be very simple or complex on the inside; we can't tell from the outside, and we don't allow ourselves to open the machine up (except when we are absolutely sure something is wrong with its design), so we are required to do things like flip the switches and read the dials to interact with the data. Once the machine is built, we don't want to have to think about how it operates.

You might think we are just making more work for ourselves, but this approach tends to do a nice job of preventing all kinds of things from going wrong.

Let's start with a example that is too simple to be of practical value, but should illustrate at least part of the concept. Your car has a tripmeter. Its job is to keep track of the distance the car has travelled since the last time its reset button was pushed. How would we model this in a programming language? In C, the tripmeter would just be a numeric variable, possibly of type `float`. The program would manipulate that variable by increasing its value in small increments, with occasional resets to zero when appropriate. What's wrong with that? A bug in the program could assign a bogus value to the variable, for any number of unexpected reasons. Anyone who has programmed in C knows what it is like to spend hours or days tracking down such a bug whose cause seems absurdly simple once it has been found. (The moment of finding the bug is commonly indicated by the sound of a loud slap to the forehead.)

The same problem would be attacked from a much different angle in an object-oriented context. The first thing a programmer asks when designing the tripmeter is not "which of the familiar data types comes closest to resembling this thing?" but "how exactly is this thing supposed to act?" The difference winds up being a profound one. It is necessary to spend a little bit of time deciding exactly what an odometer is for, and how the outside world expects to interact with it. We decide to build a little machine with controls that allow us to increment it, reset it, read its value, and nothing else.

We don't provide a way for a tripmeter to be assigned arbitrary values; why? because we all know tripmeters don't work that way. There are only a few things you should be able to do with a tripmeter, and those are all we allow. Thus, if something else in the program mistakenly tries to place some other value (say, the target temperature of the vehicle's climate control) into the tripmeter, there is an immediate indication of what went wrong. We are told when running the program (or possibly while compiling, depending on the nature of the language) that *we are not allowed to assign arbitrary values to Tripmeter objects*. The message might not be exactly that clear, but it will be reasonably close to that. It doesn't prevent the error, does it? But it quickly points us in the direction of the cause. This is only one of several ways in which OO programming can save a lot of wasted time.

We commonly take one step of abstraction above this, because it turns out to be as easy to build a factory that makes machines as it is to make an individual machine. We aren't likely to build a single tripmeter directly; rather, we arrange for any number of tripmeters to be built from a single pattern. The pattern (or if you like, the tripmeter factory) corresponds to what we call a *class*, and an individual tripmeter generated from the pattern (or made by the factory) corresponds to an *object*. Most OO languages require a class to be defined before we can have a new kind of object, but ruby does not.

It's worth noting here that the use of an OO language will not *enforce* proper OO design. Indeed it is possible in any language to write code that is unclear, sloppy, ill-conceived, buggy, and wobbly all over. What ruby does for you (as opposed, especially, to C++) is to make the practice of OO programming feel natural enough that even when you are working on a small scale you don't feel a necessity to resort to ugly code to save effort. We will be discussing the ways in which ruby accomplishes that admirable goal as this guide progresses; the next topic will be the "switches and dials" (object methods) and from there we'll move on to the "factories" (classes). Are you still with us?

Ruby User's Guide

Methods

What is a method? In OO programming, we don't think of operating on data directly from outside an object; rather, objects have some understanding of how to operate on themselves (when asked nicely to do so). You might say we pass messages to an object, and those messages will generally elicit some kind of an action or meaningful reply. This ought to happen without our necessarily knowing or caring how the object really works inside. The tasks we are allowed to ask an object to perform (or equivalently, the messages it understands) are that object's *methods.*

In ruby, we invoke a method of an object with dot notation (just as in C++ or Java). The object being talked to is named to the left of the dot.

```
ruby> "abcdef".length
   6
```

Intuitively, *this string object is being asked how long it is.* Technically, we are invoking the `length` method of the object `"abcdef"`.

Other objects may have a slightly different interpretation of `length`, or none at all. Decisions about how to respond to a message are made on the fly, during program execution, and the action taken may change depending on what a variable refers to.

```
ruby> foo = "abc"
   "abc"
ruby> foo.length
   3
ruby> foo = ["abcde", "fghij"]
   ["abcde", "fghij"]
ruby> foo.length
   2
```

What we mean by *length* can vary depending on what object we are talking about. The first time we ask `foo` for its length in the above example, it refers to a simple string, and there can only be one sensible answer. The second time, `foo` refers to an array, and we might reasonably think of its length as either 2, 5, or 10; but the most generally applicable answer is of course 2 (the other kinds of length can be figured out if wished).

```
ruby> foo[0].length
   5
ruby> foo[0].length + foo[1].length
   10
```

The thing to notice here is that an array *knows something about what it means to be an array.* Pieces of data in ruby carry such knowledge with them, so that the demands made on them can automatically be satisfied in the various appropriate ways. This relieves the programmer from the burden of memorizing a great many specific function names, because a relatively small number of method names, corresponding to concepts that we know how to express in natural language, can be applied to different kinds of data and the results will be what we expect. This feature of OO programming languages (which, IMHO, Java has done a poor job of exploiting) is called *polymorphism*.

When an object receives a message that it does not understand, an error is "raised":

```
ruby> foo = 5
   5
ruby> foo.length
ERR: (eval):1: undefined method `length' for 5(Fixnum)
```

So it is necessary to know what methods are acceptable to an object, though we need not know how the methods are processed.

If arguments are given to a method, they are generally surrounded by parentheses,

```
object.method(arg1, arg2)
```

but they can be omitted if doing so does not cause ambiguity.

```
object.method arg1, arg2
```

There is a special variable `self` in ruby; it refers to whatever object calls a method. This happens so often that for convenience the "`self.`" may be omitted from method calls from an object to itself:

```
self.method_name(args...)
```

is the same as

```
method_name(args...)
```

What we would think of traditionally as a *function call* is just this abbreviated way of writing method invocations by `self`. This makes ruby what is called a pure object oriented language. Still, functional methods behave quite similarly to the functions in other programming languages for the benefit of those who do not grok how function calls are really object methods in ruby. We can speak of *functions* as if they were not really object methods if we want to.

Ruby User's Guide

Classes

The real world is filled by objects, and we can classify them. For example, a very small child is likely to say "bow-wow" when seeing a dog, regardless of the breed; we naturally see the world in terms of these categories.

In OO programming terminology, a category of objects like "dog" is called a *class*, and some specific object belonging to a class is called an *instance* of that class.

Generally, to make an object in ruby or any other OO language, first one defines the characteristics of a class, then creates an instance. To illustrate the process, let's first define a simple Dog class.

```
ruby> class Dog
    |   def speak
    |     puts "Bow Wow"
    |   end
    | end
   nil
```

In ruby, a class definition is a region of code between the keywords `class` and `end`. A `def` inside this region begins the definition of a *method* of the class, which as we discussed in the previous chapter, corresponds to some specific behavior for objects of that class.

Now that we have defined a Dog class, we can use it to make a dog:

```
ruby> pochi = Dog.new
   #<Dog:0xbcb90>
```

We have made a new instance of the class Dog, and have given it the name `pochi`. The `new` method of any class makes a new instance. Because `pochi` is a Dog according to our class definition, it has whatever properties we decided a Dog should have. Since our idea of Dog-ness was very simple, there is just one trick we can ask `pochi` to do.

```
ruby> pochi.speak
Bow Wow
   nil
```

Making a new instance of a class is sometimes called *instantiating* that class. We need to have a dog before we can experience the pleasure of its conversation; we can't merely ask the Dog *class* to bark for us.

```
ruby> Dog.speak
ERR: (eval):1: undefined method `speak' for Dog:class
```

It makes no more sense than trying to *eat the concept of a sandwich*.

On the other hand, if we want to hear the sound of a dog without getting emotionally attached, we can create (instantiate) an ephemeral, temporary dog, and coax a little noise out of it before it disappears.

```
ruby> (Dog.new).speak   # or more commonly, Dog.new.speak
Bow Wow
   nil
```

"Wait," you say, "what's all this about the poor fellow disappearing afterwards?" It's true: if we don't bother to give it a name (as we did for `pochi`), ruby's automatic garbage collection decides it is an unwanted stray dog, and mercilessly disposes of it. Really it's okay, you know, because we can make all the dogs we want.

Ruby User's Guide

Inheritance

Our classification of objects in everyday life is naturally hierarchical. We know that *all cats are mammals*, and *all mammals are animals*. Smaller classes *inherit* characteristics from the larger classes to which they belong. If all mammals breathe, then all cats breathe.

We can express this concept in ruby:

```
ruby> class Mammal
    |   def breathe
    |     puts "inhale and exhale"
    |   end
    | end
   nil
ruby> class Cat<Mammal
    |   def speak
    |     puts "Meow"
    |   end
    | end
   nil
```

Though we didn't specify how a `Cat` should breathe, every cat will inherit that behavior from the `Mammal` class since `Cat` was defined as a subclass of `Mammal`. (In OO terminology, the smaller class is a *subclass* and the larger class is a *superclass*.) Hence from a programmer's standpoint, cats get the ability to breathe for free; after we add a `speak` method, our cats can both breathe and speak.

```
ruby> tama = Cat.new
   #<Cat:0xbd80e8>
ruby> tama.breathe
inhale and exhale
   nil
ruby> tama.speak
Meow
   nil
```

There will be situations where certain properties of the superclass should not be inherited by a particular subclass. Though birds generally know how to fly, penguins are a flightless subclass of birds.

```
ruby> class Bird
    |   def preen
    |     puts "I am cleaning my feathers."
    |   end
    |   def fly
    |     puts "I am flying."
    |   end
    | end
```

```
   nil
ruby> class Penguin<Bird
    |   def fly
    |     fail "Sorry. I'd rather swim."
    |   end
    | end
   nil
```

Rather than exhaustively define every characteristic of every new class, we need only to append or to redefine the differences between each subclass and its superclass. This use of inheritance is sometimes called *differential programming*. It is one of the benefits of object-oriented programming.

Ruby User's Guide

Redefinition of methods

In a subclass, we can change the behavior of the instances by redefining superclass methods.

```
ruby> class Human
    |   def identify
    |     puts "I'm a person."
    |   end
    |   def train_toll(age)
    |     if age < 12
    |       puts "Reduced fare.";
    |     else
    |       puts "Normal fare.";
    |     end
    |   end
    | end
   nil
ruby> Human.new.identify
I'm a person.
   nil
ruby> class Student1<Human
    |   def identify
    |     puts "I'm a student."
    |   end
    | end
   nil
ruby> Student1.new.identify
I'm a student.
   nil
```

Suppose we would rather enhance the superclass's `identify` method than entirely replace it. For this we can use `super`.

```
ruby> class Student2<Human
    |   def identify
    |     super
    |     puts "I'm a student too."
    |   end
    | end
   nil
ruby> Student2.new.identify
I'm a human.
I'm a student too.
   nil
```

`super` lets us pass arguments to the original method. It is sometimes said that there are two kinds of people...

```
ruby> class Dishonest<Human
    |   def train_toll(age)
    |     super(11) # we want a cheap fare.
```

```
         |   end
         | end
      nil
   ruby> Dishonest.new.train_toll(25)
   Reduced fare.
      nil

   ruby> class Honest<Human
         |   def train_toll(age)
         |     super(age) # pass the argument we were given
         |   end
         | end
      nil
   ruby> Honest.new.train_toll(25)
   Normal fare.
      nil
```

Ruby User's Guide

Access control

Earlier, we said that ruby has no functions, only methods. However there is more than one kind of method. In this chapter we introduce *access controls*.

Consider what happens when we define a method in the "top level", not inside a class definition. We can think of such a method as analogous to a *function* in a more traditional language like C.

```
ruby> def square(n)
    |   n * n
    | end
   nil
ruby> square(5)
   25
```

Our new method would appear not to belong to any class, but in fact ruby gives it to the `Object` class, which is a superclass of every other class. As a result, any object should now be able to use that method. That turns out to be true, but there's a small catch: it is a *private* method of every class. We'll discuss some of what this means below, but one consequence is that it may be invoked only in function style, as here:

```
ruby> class Foo
    |   def fourth_power_of(x)
    |     square(x) * square(x)
    |   end
    | end
  nil
ruby> Foo.new.fourth_power_of 10
  10000
```

We are not allowed to explicitly apply the method to an object:

```
ruby> "fish".square(5)
ERR: (eval):1: private method `square' called for
"fish":String
```

This rather cleverly preserves ruby's pure-OO nature (functions are still object methods, but the receiver is `self` implicitly) while providing functions that can be written just as in a more traditional language.

A common mental discipline in OO programming, which we have hinted at in an earlier chapter, concerns the separation of *specification* and *implementation*, or *what* tasks an object is supposed to accomplish and *how* it actually accomplishes them. The internal workings of an object should be kept generally hidden from its users; they should only care about what goes in and what comes out, and trust the object to know what it is doing internally. As such it is often helpful for classes to have methods that the outside world does not see, but which are used internally (and can be improved by the programmer whenever desired, without changing the way users see objects of that class). In the trivial example below, think of `engine` as the invisible inner workings of the class.

```
ruby> class Test
    |   def times_two(a)
    |     puts "#{a} times two is #{engine(a)}"
    |   end
    |   def engine(b)
    |     b*2
    |   end
    |   private:engine  # this hides engine from users
    | end
   Test
ruby> test = Test.new
   #<Test:0x4017181c>
ruby> test.engine(6)
ERR: (eval):1: private method `engine' called for
#<Test:0x4017181c>
ruby> test.times_two(6)
6 times two is 12.
   nil
```

We might have expected `test.engine(6)` to return 12, but instead we learn that `engine` is inaccessible when we are acting as a user of a `Test` object. Only other `Test` methods, such as `times_two`, are allowed to use `engine`. We are required to go through the public interface, which consists of the `times_two` method. The programmer who is in charge of this class can change `engine` freely (here, perhaps by changing `b*2` to `b+b`, assuming for the sake of argument that it improved performance) without affecting how the user interacts with `Test` objects. This example is of course much too simple to be useful; the benefits of access controls become more clear only when we begin to create more complicated and interesting classes.

Ruby User's Guide

Singleton methods

The behavior of an instance is determined by its class, but there may be times we know that a particular instance should have special behavior. In most languages, we must go to the trouble of defining another class, which would then only be instantiated once. In ruby we can give any object its own methods.

```
ruby> class SingletonTest
    |   def size
    |     25
    |   end
    | end
   nil
ruby> test1 = SingletonTest.new
   #<SingletonTest:0xbc468>
ruby> test2 = SingletonTest.new
   #<SingletonTest:0xbae20>
ruby> def test2.size
    |    10
    | end
   nil
ruby> test1.size
   25
ruby> test2.size
   10
```

In this example, `test1` and `test2` belong to same class, but `test2` has been given a redefined `size` method and so they behave differently. A method given only to a single object is called a *singleton method*.

Singleton methods are often used for elements of a graphic user interface (GUI), where different actions need to be taken when different buttons are pressed.

Singleton methods are not unique to ruby, as they appear in CLOS, Dylan, etc. Also, some languages, for example, Self and NewtonScript, have singleton methods only. These are sometimes called *prototype-based* languages.

Ruby User's Guide

Modules

Modules in ruby are similar to classes, except:

- A module can have no instances.
- A module can have no subclasses.
- A module is defined by `module ... end`.

Actually... the Module class of module is the superclass of the Class class of class. Got that? No? Let's move on.

There are two typical uses of modules. One is to collect related methods and constants in a central location. The `Math` module in ruby's standard library plays such a role:

```
ruby> Math.sqrt(2)
   1.41421
ruby> Math::PI
   3.14159
```

The `::` operator tells the ruby interpreter which module it should consult for the value of a constant (conceivably, some module besides `Math` might mean something else by `PI`). If we want to refer to the methods or constants of a module directly without using `::`, we can `include` that module:

```
ruby> include Math
   Object
ruby> sqrt(2)
   1.41421
ruby> PI
   3.14159
```

Another use of modules is called *mixin*. Some OO programming langages, including C++, allow *multiple inheritance*, that is, inheritance from more than one superclass. A real-world example of multiple inheritance is an alarm clock; you can think of alarm clocks as belonging to the class of *clocks* and also the class of *things with buzzers*.

Ruby purposely does not implement true multiple inheritance, but the *mixin* technique is a good alternative. Remember that modules cannot be instantiated or subclassed; but if we `include` a module in a class definition, its methods are effectively appended, or "mixed in", to the class.

Mixin can be thought of as a way of asking for whatever particular properties we want to have. For example, if a class has a working `each` method, mixing in the standard library's `Enumerable` module gives us `sort` and `find` methods for free.

This use of modules gives us the basic functionality of multiple inheritance but allows us to represent class relationships with a simple tree structure, and so simplifies the language implementation considerably (a similar choice was made by the designers of Java).

Ruby User's Guide

Procedure objects

It is often desirable to be able to specify responses to unexpected events. As it turns out, this is most easily done if we can pass blocks of code as arguments to other methods, which means we want to be able to treat code as if it were data.

A new *procedure object* is formed using `proc`:

```
ruby> quux = proc {
    |   puts "QUUXQUUXQUUX!!!"
    | }
   #<Proc:0x4017357c>
```

Now what `quux` refers to is an object, and like most objects, it has behavior that can be invoked. Specifically, we can ask it to execute, via its `call` method:

```
ruby> quux.call
QUUXQUUXQUUX!!!
   nil
```

So, after all that, can `quux` be used as a method argument? Sure.

```
ruby> def run( p )
    |   puts "About to call a procedure..."
    |   p.call
    |   puts "There: finished."
    | end
   nil
ruby> run quux
About to call a procedure...
QUUXQUUXQUUX!!!
There: finished.
   nil
```

The `trap` method lets us assign the response of our choice to any system signal.

```
ruby> inthandler = proc{ puts "^C was pressed." }
   #<Proc:0x401730a4>
ruby> trap "SIGINT", inthandler
   #<Proc:0x401735e0>
```

Normally pressing *^C* makes the interpreter quit. Now a message is printed and the interpreter continues running, so you don't lose the work you were doing. (You're not trapped in the interpreter forever; you can still exit by typing `exit`.)

A final note before we move on to other topics: it's not strictly necessary to give a procedure object a name before binding it to a signal. An equivalent *anonymous* procedure object would look like

```
ruby> trap "SIGINT", proc{ puts "^C was pressed." }
   nil
```

or more compactly still,

```
ruby> trap "SIGINT", 'puts "^C was pressed."'
   nil
```

This abbreviated form provides some convenience and readability when you write small anonymous procedures.

Ruby User's Guide

Variables

Ruby has three kinds of variables, one kind of constant and exactly two pseudo-variables. The variables and the constants have no type. While untyped variables have some drawbacks, they have many more advantages and fit well with ruby's *quick and easy* philosophy.

Variables must be declared in most languages in order to specify their type, modifiability (i.e., whether they are constants), and scope; since type is not an issue, and the rest is evident from the variable name as you are about to see, we do not need variable declarations in ruby.

The first character of an identifier categorizes it at a glance:

`$`	global variable
`@`	instance variable
`[a-z]` or `_`	local variable
`[A-Z]`	constant

The only exceptions to the above are ruby's pseudo-variables: `self`, which always refers to the currently executing object, and `nil`, which is the meaningless value assigned to uninitialized variables. Both are named as if they are local variables, but `self` is a global variable maintained by the interpreter, and `nil` is really a constant. As these are the only two exceptions, they don't confuse things too much.

You man not assign values to `self` or `nil`. `main`, as a value of `self`, refers to the top-level object:

```
ruby> self
   main
ruby> nil
   nil
```

Ruby User's Guide

Global variables

A global variable has a name beginning with $. It can be referred to from anywhere in a program. Before initialization, a global variable has the special value nil.

```
ruby> $foo
   nil
ruby> $foo = 5
   5
ruby> $foo
   5
```

Global variables should be used sparingly. They are dangerous because they can be written to from anywhere. Overuse of globals can make isolating bugs difficult; it also tends to indicate that the design of a program has not been carefully thought out. Whenever you do find it necessary to use a global variable, be sure to give it a descriptive name that is unlikely to be inadvertently used for something else later (calling it something like $foo as above is probably a bad idea).

One nice feature of a global variable is that it can be traced; you can specify a procedure which is invoked whenever the value of the variable is changed.

```
ruby> trace_var :$x, proc{puts "$x is now #{$x}"}
   nil
ruby> $x = 5
$x is now 5
   5
```

When a global variable has been rigged to work as a trigger to invoke a procedure whenever changed, we sometimes call it an *active variable*. For instance, it might be useful for keeping a GUI display up to date.

There is a collection of special variables whose names consist of a dollar sign ($) followed by a single character. For example, $$ contains the process id of the ruby interpreter, and is read-only. Here are the major system variables:

$!	latest error message
$@	location of error
$_	string last read by gets
$.	line *number* last read by interpreter
$&	string last matched by regexp
$~	the last regexp match, as an array of subexpressions
$*n*	the *nth* subexpression in the last match (same as $~[*n*])
$=	case-insensitivity flag
$/	input record separator

$\	output record separator
$0	the name of the ruby script file
$*	the command line arguments
$$	interpreter's process ID
$?	exit status of last executed child process

In the above, **$_** and **$~** have local scope. Their names suggest they should be global, but they are much more useful this way, and there are historical reasons for using these names.

Ruby User's Guide

Instance variables

An instance variable has a name beginning with `@`, and its scope is confined to whatever object `self` refers to. Two different objects, even if they belong to the same class, are allowed to have different values for their instance variables. From outside the object, instance variables cannot be altered or even observed (i.e., ruby's instance variables are never *public*) except by whatever methods are explicitly provided by the programmer. As with globals, instance variables have the `nil` value until they are initialized.

Instance variables of ruby do not need declaration. This implies a flexible structure of objects. In fact, each instance variable is dynamically appended to an object when it is first referenced.

```
ruby> class InstTest
    |   def set_foo(n)
    |     @foo = n
    |   end
    |   def set_bar(n)
    |     @bar = n
    |   end
    | end
   nil
ruby> i = InstTest.new
   #<InstTest:0x83678>
ruby> i.set_foo(2)
   2
ruby> i
   #<InstTest:0x83678 @foo=2>
ruby> i.set_bar(4)
   4
ruby> i
   #<InstTest:0x83678 @foo=2, @bar=4>
```

Notice above that `i` does not report a value for `@bar` until after the `set_bar` method is invoked.

Ruby User's Guide

Local variables

A local variable has a name starting with a lower case letter or an underscore character (_). Local variables do not, like globals and instance variables, have the value nil before initialization:

```
ruby> $foo
   nil
ruby> @foo
   nil
ruby> foo
ERR: (eval):1: undefined local variable or method `foo' for
main(Object)
```

The first assignment you make to a local variable acts something like a declaration. If you refer to an uninitialized local variable, the ruby interpreter cannot be sure whether you are referencing a bogus variable or calling a nonexistent method; hence the error message you see above.

Generally, the scope of a local variable is one of

- proc{ ... }
- loop{ ... }
- def ... end
- class ... end
- module ... end
- the entire program (unless one of the above applies)

In the next example, defined? is an operator which checks whether an identifier is defined. It returns a description of the identifier if it is defined, or nil otherwise. As you see, bar's scope is local to the loop; when the loop exits, bar is undefined.

```
ruby> foo = 44; puts foo; defined?(foo)
44
   "local-variable"
ruby> loop{bar=45; puts bar; break}; defined?(bar)
45
   nil
```

Procedure objects that live in the same scope share whatever local variables also belong to that scope. Here, the local variable bar is shared by main and the procedure objects p1 and p2:

```
ruby> bar=nil
   nil
ruby> p1 = proc{|n| bar=n}
   #<Proc:0x8deb0>
ruby> p2 = proc{bar}
```

```
   #<Proc:0x8dce8>
ruby> p1.call(5)
   5
ruby> bar
   5
ruby> p2.call
   5
```

Note that the "bar=nil" at the beginning cannot be omitted; it ensures that the scope of bar will encompass p1 and p2. Otherwise p1 and p2 would each end up with its own local variable bar, and calling p2 would have resulted in an "undefined local variable or method" error. We could have said bar=0 instead, but using nil is a courtesy to others who will read your code later. It indicates fairly clearly that you are only establishing scope, because the value being assigned is not intended to be meaningful.

A powerful feature of procedure objects follows from their ability to be passed as arguments: shared local variables remain valid even when they are passed out of the original scope.

```
ruby> def box
    |   contents = nil
    |   get = proc{contents}
    |   set = proc{|n| contents = n}
    |   return get, set
    | end
   nil
ruby> reader, writer = box
   [#<Proc:0x40170fc0>, #<Proc:0x40170fac>]
ruby> reader.call
   nil
ruby> writer.call(2)
   2
ruby> reader.call
   2
```

Ruby is particularly smart about scope. It is evident in our example that the contents variable is being shared between the reader and writer. But we can also manufacture multiple reader-writer pairs using box as defined above; each pair shares a contents variable, and the pairs do not interfere with each other.

```
ruby> reader_1, writer_1 = box
   [#<Proc:0x40172820>, #<Proc:0x4017280c>]
ruby> reader_2, writer_2 = box
   [#<Proc:0x40172668>, #<Proc:0x40172654>]
ruby> writer_1.call(99)
   99
ruby> reader_1.call
   99
ruby> reader_2.call  # nothing is in this box yet
   nil
```

This kind of programming could be considered a perverse little object-oriented framework. The box method acts something like a class, with get and set serving as methods (except those aren't really the method *names*, which could vary with each box instance) and `contents` being the lone instance variable. Of course, using ruby's legitimate class framework leads to much more readable code.

Ruby User's Guide

Class constants

A constant has a name starting with an uppercase character. It should be assigned a value at most once. In the current implementation of ruby, reassignment of a constant generates a warning but not an error (the non-ANSI version of eval.rb does not report the warning):

```
ruby>fluid=30
   30
ruby>fluid=31
   31
ruby>Solid=32
   32
ruby>Solid=33
   (eval):1: warning: already initialized constant Solid
   33
```

Constants may be defined within classes, but unlike instance variables, they are accessible outside the class.

```
ruby> class ConstClass
    |   C1=101
    |   C2=102
    |   C3=103
    |   def show
    |     puts "#{C1} #{C2} #{C3}"
    |   end
    | end
   nil
ruby> C1
ERR: (eval):1: uninitialized constant C1
ruby> ConstClass::C1
   101
ruby> ConstClass.new.show
101 102 103
   nil
```

Constants can also be defined in modules.

```
ruby> module ConstModule
    |   C1=101
    |   C2=102
    |   C3=103
    |   def showConstants
    |     puts "#{C1} #{C2} #{C3}"
    |   end
    | end
   nil
```

```
ruby> C1
ERR: (eval):1: uninitialized constant C1
ruby> include ConstModule
   Object
ruby> C1
   101
ruby> showConstants
101 102 103
   nil
ruby> C1=99  # not really a good idea
   99
ruby> C1
   99
ruby> ConstModule::C1
   101
ruby> ConstModule::C1=99   # .. this was not allowed in
earlier versions
   (eval):1: warning: already initialized constant C1
   99
ruby> ConstModule::C1  # "enough rope to shoot yourself in
the foot"
   99
```

Ruby User's Guide

Exception processing: rescue

An executing program can run into unexpected problems. A file that it it wants to read might not exist; the disk might be full when it wants to save some data; the user may provide it with some unsuitable kind of input.

```
ruby> file = open("some_file")
ERR: (eval):1:in `open': No such file or directory -
some_file
```

A robust program will handle these situations sensibly and gracefully. Meeting that expectation can be an exasperating task. C programmers are expected to check the result of every system call that could possibly fail, and immediately decide what is to be done:

```
FILE *file = fopen("some_file", "r");
if (file == NULL) {
  fprintf( stderr, "File doesn't exist.\n" );
  exit(1);
}
bytes_read = fread( buf, 1, bytes_desired, file );
if (bytes_read != bytes_desired ) {
  /* do more error handling here ... */
}
...
```

This is such a tiresome practice that programmers can tend to grow careless and neglect it, and the result is a program that doesn't handle exceptions well. On the other hand, doing the job right can make programs hard to read, because there is so much error handling cluttering up the meaningful code.

In ruby, as in many modern languages, we can handle exceptions for blocks of code in a compartmentalized way, thus dealing with surprises effectively but not unduly burdening either the programmer or anyone else trying to read the code later. The block of code marked with `begin` executes until there is an exception, which causes control to be transferred to a block of error handling code, which is marked with `rescue`. If no exception occurs, the `rescue` code is not used. The following method returns the first line of a text file, or `nil` if there is an exception:

```
def first_line( filename )
  begin
    file = open("some_file")
    info = file.gets
    file.close
    info  # Last thing evaluated is the return value
  rescue
    nil   # Can't read the file? then don't return a string
  end
end
```

There will be times when we would like to be able to creatively work around a problem. Here, if the file we want is unavailable, we try to use standard input instead:

```
begin
  file = open("some_file")
rescue
  file = STDIN
end

begin
  # ... process the input ...
rescue
  # ... and deal with any other exceptions here.
end
```

`retry` can be used in the `rescue` code to start the `begin` code over again. It lets us rewrite the previous example a little more compactly:

```
fname = "some_file"
begin
  file = open(fname)
  # ... process the input ...
rescue
  fname = "STDIN"
  retry
end
```

However, there is a flaw here. A nonexistent file will make this code retry in an infinite loop. You need to watch out for such pitfalls when using `retry` for exception processing.

Every ruby library raises an exception if any error occurs, and you can raise exceptions explicitly in your code too. To raise an exception, use `raise`. It takes one argument, which should be a string that describes the exception. The argument is optional but should not be omitted. It can be accessed later via the special global variable `$!`.

```
ruby> raise "test error"
   test error
ruby> begin
    |   raise "test2"
    | rescue
    |   puts "An error occurred: #{$!}"
    | end
An error occurred: test2
   nil
```

Ruby User's Guide

Exception processing: ensure

There may be cleanup work that is necessary when a method finishes its work. Perhaps an open file should be closed, buffered data should be flushed, etc. If there were always only one exit point for each method, we could confidently put our cleanup code in one place and know that it would be executed; however, a method might return from several places, or our intended cleanup code might be unexpectedly skipped because of an exception.

```
begin
  file = open("/tmp/some_file", "w")
  # ... write to the file ...
  file.close
end
```

In the above, if an exception occurred during the section of code where we were writing to the file, the file would be left open. And we don't want to resort to this kind of redundancy:

```
begin
  file = open("/tmp/some_file", "w")
  # ... write to the file ...
  file.close
rescue
  file.close
  fail # raise an exception
end
```

It's clumsy, and gets out of hand when the code gets more complicated because we have to deal with every `return` and `break`.

For this reason we add another keyword to the "`begin...rescue...end`" scheme, which is `ensure`. The `ensure` code block executes regardless of the success or failure of the `begin` block.

```
begin
  file = open("/tmp/some_file", "w")
  # ... write to the file ...
rescue
  # ... handle the exceptions ...
ensure
  file.close   # ... and this always happens.
end
```

It is possible to use `ensure` without `rescue`, or vice versa, but if they are used together in the same `begin...end` block, the `rescue` must precede the `ensure`.

Ruby User's Guide

Accessors

What is an accessor?

We briefly discussed instance variables in an earlier chapter, but haven't done much with them yet. An object's instance variables are its attributes, the things that generally distinguish it from other objects of the same class. It is important to be able to write and read these attributes; doing so requires methods called *attribute accessors*. We'll see in a moment that we don't always have to write accessor methods explicitly, but let's go through all the motions for now. The two kinds of accessors are *writers* and *readers*.

```
ruby> class Fruit
    |   def set_kind(k)  # a writer
    |     @kind = k
    |   end
    |   def get_kind     # a reader
    |     @kind
    |   end
    | end
   nil
ruby> f1 = Fruit.new
   #<Fruit:0xfd7e7c8c>
ruby> f1.set_kind("peach")  # use the writer
   "peach"
ruby> f1.get_kind           # use the reader
   "peach"
ruby> f1                    # inspect the object
   #<Fruit:0xfd7e7c8c @kind="peach">
```

Simple enough; we can store and retrieve information about what kind of fruit we're looking at. But our method names are a little wordy. The following is more concise, and more conventional:

```
ruby> class Fruit
    |   def kind=(k)
    |     @kind = k
    |   end
    |   def kind
    |     @kind
    |   end
    | end
   nil
ruby> f2 = Fruit.new
   #<Fruit:0xfd7e7c8c>
ruby> f2.kind = "banana"
   "banana"
ruby> f2.kind
   "banana"
```

The `inspect` method

A short digression is in order. You've noticed by now that when we try to look at an object directly, we are shown something cryptic like `#<anObject:0x83678>`. This is just a default behavior, and we are free to change it. All we need to do is add a method named `inspect`. It should return a string that describes the object in some sensible way, including the states of some or all of its instance variables.

```
ruby> class Fruit
    |   def inspect
    |     "a fruit of the #{@kind} variety"
    |   end
    | end
   nil
ruby> f2
   "a fruit of the banana variety"
```

A related method is `to_s` (convert to string), which is used when printing an object. In general, you can think of `inspect` as a tool for when you are writing and debugging programs, and `to_s` as a way of refining program output. `eval.rb` uses `inspect` whenever it displays results. You can use the `p` method to easily get debugging output from programs.

```
# These two lines are equivalent:
p anObject
puts anObject.inspect
```

Making accessors the easy way

Since many instance variables need accessor methods, Ruby provides convenient shortcuts for the standard forms.

Shortcut	*Effect*
`attr_reader :v`	`def v; @v; end`
`attr_writer :v`	`def v=(value); @v=value; end`
`attr_accessor :v`	`attr_reader :v; attr_writer :v`
`attr_accessor :v, :w`	`attr_accessor :v; attr_accessor :w`

Let's take advantage of this and add freshness information. First we ask for an automatically generated reader and writer, and then we incorporate the new information into `inspect`:

```
ruby> class Fruit
    |   attr_accessor :condition
    |   def inspect
    |     "a #{@condition} #{@kind}"
    |   end
    | end
   nil
ruby> f2.condition = "ripe"
   "ripe"
ruby> f2
   "a ripe banana"
```

More fun with fruit

If nobody eats our ripe fruit, perhaps we should let time take its toll.

```
ruby> class Fruit
    |   def time_passes
    |     @condition = "rotting"
    |   end
    | end
   nil
ruby> f2
   "a ripe banana"
ruby> f2.time_passes
   "rotting"
ruby> f2
   "a rotting banana"
```

But while playing around here, we have introduced a small problem. What happens if we try to create a third piece of fruit now? Remember that instance variables don't exist until values are assigned to them.

```
ruby> f3 = Fruit.new
ERR: failed to convert nil into String
```

It is the `inspect` method that is complaining here, and with good reason. We have asked it to report on the kind and condition of a piece of fruit, but as yet `f3` has not been assigned either attribute. If we wanted to, we could rewrite the `inspect` method so it tests instance variables using the `defined?` method and then only reports on them if they exist, but maybe that's not very useful; since every piece of fruit has a kind and condition, it seems we should make sure those always get defined somehow. That is the topic of the next chapter.

Ruby User's Guide

Object initialization

Our Fruit class from the previous chapter had two instance variables, one to describe the kind of fruit and another to describe its condition. It was only after writing a custom `inspect` method for the class that we realized it didn't make sense for a piece of fruit to lack those characteristics. Fortunately, ruby provides a way to ensure that instance variables always get initialized.

The `initialize` method

Whenever Ruby creates a new object, it looks for a method named `initialize` and executes it. So one simple thing we can do is use an `initialize` method to put default values into all the instance variables, so the `inspect` method will have something to say.

```
ruby> class Fruit
    |   def initialize
    |     @kind = "apple"
    |     @condition = "ripe"
    |   end
    | end
   nil
ruby> f4 = Fruit.new
   "a ripe apple"
```

Changing assumptions to requirements

There will be times when a default value doesn't make a lot of sense. Is there such a thing as a default kind of fruit? It may be preferable to require that each piece of fruit have its kind specified at the time of its creation. To do this, we would add a formal argument to the `initialize` method. For reasons we won't get into here, arguments you supply to `new` are actually delivered to `initialize`.

```
ruby> class Fruit
    |   def initialize( k )
    |     @kind = k
    |     @condition = "ripe"
    |   end
    | end
   nil
ruby> f5 = Fruit.new "mango"
   "a ripe mango"
ruby> f6 = Fruit.new
ERR: (eval):1:in `initialize': wrong # of arguments(0 for 1)
```

Flexible initialization

Above we see that once an argument is associated with the `initialize` method, it can't be left off without generating an error. If we want to be more considerate, we can use the argument if it is given, or fall back to default values otherwise.

```
ruby> class Fruit
    |   def initialize( k="apple" )
    |     @kind = k
    |     @condition = "ripe"
    |   end
    | end
   nil
ruby> f5 = Fruit.new "mango"
   "a ripe mango"
ruby> f6 = Fruit.new
   "a ripe apple"
```

You can use default argument values for any method, not just `initialize`. The argument list must be arranged so that those with default values come last.

Sometimes it is useful to provide several ways to initialize an object. Although it is outside the scope of this tutorial, ruby supports object reflection and variable-length argument lists, which together effectively allow method overloading.

Ruby User's Guide

Nuts and bolts

This chapter addresses a few practical issues.

Statement delimiters

Some languages require some kind of punctuation, often a semicolon (`;`), to end each statement in a program. Ruby instead follows the convention used in shells like `sh` and `csh`. Multiple statements on one line must be separated by semicolons, but they are not required at the end of a line; a linefeed is treated like a semicolon. If a line ends with a backslash (`\`), the linefeed following it is ignored; this allows you to have a single logical line that spans several lines.

Comments

Why write comments? Although well written code tends to be self-documenting, it is often helpful to scribble in the margins, and it can be a mistake to believe that others will be able to look at your code and immediately see it the way you do. Besides, for practical purposes, you yourself are a different person within a few days anyway; which of us hasn't gone back to fix or enhance a program after the passage of time and said, I know I wrote this, but what in blazes does it mean?

Some experienced programmers will point out, quite correctly, that contradictory or outdated comments can be worse than none at all. Certainly, comments shouldn't be a substitute for readable code; if your code is unclear, it's probably also buggy. You may find that you need to comment more while you are learning ruby, and then less as you become better at expressing your ideas in simple, elegant, readable code.

Ruby follows a common scripting convention, which is to use a pound symbol (`#`) to denote the start of a comment. Anything following an unquoted `#`, to the end of the line on which it appears, is ignored by the interpreter.

Also, to facilitate large comment blocks, the ruby interpreter also ignores anything between a line starting with "`=begin`" and another line starting with "`=end`".

```
#!/usr/bin/env ruby

=begin
**********************************************************************
  This is a comment block, something you write for the benefit
of
  human readers (including yourself).  The interpreter ignores
it.
  There is no need for a '#' at the start of every line.
**********************************************************************
=end
```

Organizing your code

Ruby's unusually high level of dynamism means that classes, modules, and methods exist only after their defining code runs. If you're used to programming in a more static language, this can sometimes lead to surprises.

```
# The below results in an "undefined method" error:

puts successor(3)

def successor(x)
  x + 1
end
```

Although the interpreter checks over the entire script file for syntax before executing it, the `def successor ... end` code has to actually run in order to create the `successor` method. So the order in which you arrange a script can matter.

This does not, as it might seem at first glance, force you to organize your code in a strictly bottom-up fashion. When the interpreter encounters a method definition, it can safely include undefined references, as long as you can be sure they will be defined by the time the method is actually invoked:

```
# Conversion of fahrenheit to celsius, broken
# down into two steps.

def f_to_c(f)
  scale(f - 32.0)  # This is a forward reference, but it's
okay.
end

def scale(x)
  x * 5.0 / 9.0
end

printf "%.1f is a comfortable temperature.\n", f_to_c(72.3)
```

So while this may seem less convenient than what you may be used to in Perl or Java, it is less restrictive than trying to write C without prototypes (which would require you to always maintain a partial ordering of what references what). Putting top-level code at the bottom of a source file always works. And even this is less of an annoyance than it might at first seem. A sensible and painless way to enforce the behavior you want is to define a `main` function at the top of the file, and call it from the bottom.

```
#!/usr/bin/env ruby

def main
  # Express the top level logic here...
end

# ... put support code here, organized as you see fit ...

main # ... and start execution here.
```

It also helps that ruby provides tools for breaking complicated programs into readable, reusable, logically related chunks. We have already seen the use of `include` for accessing modules. You will also find the `load` and `require` facilities useful. `load` works as if the file it refers to were copied and pasted in (something like the `#include` preprocessor directive in C). `require` is somewhat more sophisticated, causing code to be loaded at most once and only when needed.

That's it...

This tutorial should be enough to get you started writing programs in Ruby. As further questions arise, you can get more help from the user community, and from an always-growing body of printed and online resources.

Good luck, and happy coding!