Table of Contents

Ruby in a Nutshell By Yukihiro Matsumoto ISBN: 0-59600-214-9 Publisher: O'Reilly
Print Publication Date: 2001/11/01

Chapter 2. Language Basics

Ruby does what you'd expect it to do. It is highly consistent, and allows you to get down to work without having to worry about the language itself getting in your way.

2.1. Command-Line Options

Like most scripting language interpreters, Ruby is generally run from the command line. The interpreter can be invoked with the following options, which control the environment and behavior of the interpreter itself:

```
ruby [ options ] [—] [ programfile ] [ argument... ]
```

`-a`

Used with `-n` or `-p` to split each line. Split output is stored in `$F`.

`-c`

Checks syntax only, without executing program.

`-C dir`

Changes directory before executing (equivalent to `-X`).

`-d`

Enables debug mode (equivalent to `-debug`). Sets `$DEBUG` to `true`.

`-e` *`prog`*

Specifies *`prog`* as the program from the command line. Specify multiple `-e` options for multiline programs.

`-F` *`pat`*

Specifies *`pat`* as the default separator pattern (`$;`) used by `split`.

`-h`

Displays an overview of command-line options (equivalent to `-help`).

`-i [` *`ext`*`]`

Overwrites the file contents with program output. The original file is saved with the extension *`ext`*. If *`ext`* isn't specified, the original file is deleted.

`-I` *`dir`*

Adds *`dir`* as the directory for loading libraries.

`-K [` *`kcode`*`]`

Specifies the multibyte character set code (`e` or `E` for EUC (extended Unix code); `s` or `S` for SJIS (Shift-JIS); `u` or `U` for UTF-8; and `a`, A, `n`, or `N` for ASCII).

`-l`

Enables automatic line-end processing. Chops a newline from input lines and appends a newline to output lines.

`-n`

Places code within an input loop (as in `while gets; ... end`).

`-0[` *`octal`*`]`

Sets default record separator (`$/`) as an octal. Defaults to `\0` if *`octal`* not specified.

`-p`

Places code within an input loop. Writes `$_` for each iteration.

`-r` *`lib`*

Uses `require` to load *`lib`* as a library before executing.

`-s`

Interprets any arguments between the program name and filename arguments fitting the pattern `-`*`xxx`* as a switch and defines the corresponding variable.

`$xxx`.`-S`

Searches for a program using the environment variable `PATH`.

`-T [level]`

Sets the level for tainting checks (1 if level not specified). Sets the `$SAFE` variable.

`-v`

Displays version and enables `verbose` mode (equivalent to `--verbose`).

`-w`

Enables verbose mode. If `programfile` not specified, reads from `STDIN`.

`-x [ dir]`

Strips text before `#!ruby` line. Changes directory to *`dir`* before executing if *`dir`* is specified.

`-X dir`

Changes directory before executing (equivalent to `-c`).

`-y`

Enables parser debug mode (equivalent to `--yydebug`).

`--copyright`

Displays copyright notice.

`--debug`

Enables debug mode (equivalent to `-d`).

`--help`

Displays an overview of command-line options (equivalent to `-h`).

`--version`

Displays version.

`--verbose`

Enables verbose mode (equivalent to `-v`). Sets `$VERBOSE` to `true`.

`--yydebug`

Enables parser debug mode (equivalent to `-y`).

Single character command-line options can be combined. The following two lines express the same meaning:

```
ruby -ne 'print if /Ruby/' /usr/share/dict/words
ruby -n -e 'print if /Ruby/' /usr/share/dict/words
```

2.2. Environment Variables

In addition to using arguments and options on the command line, the Ruby interpreter uses the following environment variables to control its behavior. The `ENV` object contains a list of current environment variables.

`DLN_LIBRARY_PATH`

Search path for dynamically loaded modules.

`HOME`

Directory moved to when no argument is passed to `Dir::chdir`. Also used by `File::expand_path` to expand "~".

`LOGDIR`

Directory moved to when no arguments are passed to `Dir::chdir` and environment variable `HOME` isn't set.

`PATH`

Search path for executing subprocesses and searching for Ruby programs with the `-S` option. Separate each path with a colon (semicolon in DOS and Windows).

Ruby in a Nutshell By Yukihiro Matsumoto ISBN: 0-59600-214-9 Publisher: O'Reilly
Print Publication Date: 2001/11/01

`RUBYLIB`

Search path for libraries. Separate each path with a colon (semicolon in DOS and Windows).

`RUBYLIB_PREFIX`

Used to modify the RUBYLIB search path by replacing prefix of library *`path1`* with *`path2`* using the format *`path1`*;*`path2`* or *`path1path2`*. For example, if `RUBYLIB` is:

```
/usr/local/lib/ruby/site_ruby
```

and `RUBYLIB_PREFIX` is:

```
/usr/local/lib/ruby;f:/ruby
```

Ruby searches `f:/ruby/site_ruby`. Works only with DOS, Windows, and OS/2 versions.

`RUBYOPT`

Command-line options passed to Ruby interpreter. Ignored in taint mode (where `$SAFE` is greater than 0).

`RUBYPATH`

With `-S` option, search path for Ruby programs. Takes precedence over `PATH`. Ignored in taint mode (where `$SAFE` is greater than 0).

`RUBYSHELL`

Specifies shell for spawned processes. If not set, `SHELL` or `COMSPEC` are checked.

2.3. Lexical Conventions

Ruby programs are composed of elements already familiar to most programmers: lines, whitespace, comments, identifiers, reserved words, literals, etc. Particularly for those programmers coming from other scripting languages such as Perl, Python or tcl, you'll find Ruby's conventions familiar, or at least straightforward enough not to cause much trouble.

2.3.1. Whitespace

We'll leave the thorny questions like "How much whitespace makes code more readable and how much is distracting?" for another day. If you haven't already caught onto this theme, the Ruby interpreter will do pretty much what you expect with respect to whitespace in your code.

Whitespace characters such as spaces and tabs are generally ignored in Ruby code, except when they appear in strings. Sometimes, however, they are used to interpret ambiguous statements. Interpretations of this sort produce warnings when the `-w` option is enabled.

`a + b`

Interpreted as `a+b` (`a` is a local variable)

`a  +b`

Interpreted as `a(+b)` (`a`, in this case, is a method call)

2.3.2. Line Endings

Ruby interprets semicolons and newline characters as the ending of a statement. However, if Ruby encounters operators, such as `+`, `-`, or backslash at the end of a line, they indicate the continuation of a statement.

2.3.3. Comments

Comments are lines of annotation within Ruby code that are ignored at runtime. Comments extend from `#` to the end of the line.

```
# This is a comment.
```

Ruby code can contain embedded documents too. Embedded documents extend from a line beginning with `=begin` to the next line beginning with `=end`. `=begin` and `=end` must come at the beginning of a line.

```
=begin
This is an embedded document.
=end
```

2.3.4. Identifiers

Identifiers are names of variables, constants, and methods. Ruby distinguishes between identifiers consisting of uppercase characters and those of lowercase characters. Identifier names may consist of alphanumeric characters and the underscore character (_). You can distinguish a variable's type by the initial character of its identifier.

2.3.5. Reserved Words

The following list shows the reserved words in Ruby:

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

These reserved words may not be used as constant or local variable names. They can, however, be used as method names if a receiver is specified.

2.4. Literals

I've often wondered why we programmers are so enamored with literals. I'm waiting for the day when a language comes along and introduces "figuratives." In the interim, the rules Ruby uses for literals are simple and intuitive, as you'll see the following sections.

2.4.1. Numbers

Strings and numbers are the bread and butter of literals. Ruby provides support for both integers and floating-point numbers, using classes `Fixnum`, `Bignum`, and `Float`.

2.4.1.1. Integers

Integers are instances of class `Fixnum` or `Bignum`:

```
123                     # decimal
1_234                   # decimal with underline
0377                    # octal
0xff                    # hexadecimal
0b1011                  # binary
?a                      # character code for 'a'
12345678901234567890    # Bignum:  an integer of infinite length
```

2.4.1.2. Floating-point numbers

Floating-point numbers are instances of class `Float`:

```
123.4                     # floating point value
1.0e6                     # scientific notation
4E20                      # dot not required
4e+20                     # sign before exponential
```

2.4.2. Strings

A string is an array of bytes (octets) and an instance of class `String`:

`"abc"`

Double-quoted strings allow substitution and backslash notation.

`'abc'`

Single-quoted strings don't allow substitution and allow backslash notation only for `\\` and `\'`.

2.4.2.1. String concatenation

Adjacent strings are concatenated at the same time Ruby parses the program.

```
"foo" "bar"           # means "foobar"
```

2.4.2.2. Expression substitution

`#$var` and `#@var` are abbreviated forms of `#{$var}` and `#{@var}`. Embeds value of expression in `#{...}` into a string.

2.4.2.3. Backslash notation

In double-quoted strings, regular expression literals, and command output, backslash notation can be represent unprintable characters, as shown in Table 2-1.

Table 2-1. Backslash notations

Sequence	Character represented
`\n`	Newline (0x0a)
`\r`	Carriage return (0x0d)
`\f`	Formfeed (0x0c)
`\b`	Backspace (0x08)
`\a`	Bell (0x07)
`\e`	Escape (0x1b)
`\s`	Space (0x20)
`\nnn`	Octal notation (*n* being 0-7)
`\xnn`	Hexadecimal notation (*n* being 0-9, a-f, or A-F)
`\cx, \C-x`	Control-*x*
`\M-x`	Meta-x (*c* | 0x80)
`\M-\C-x`	Meta-Control-*x*
`\x`	Character *x*

`` `command` ``

Converts command output to a string. Allows substitution and backslash notation

2.4.2.4. General delimited strings

The delimiter `!` in expressions like this: `%q!...!` can be an arbitrary character. If the delimiter is any of the following: `( [ { <`, the end delimiter becomes the corresponding closing delimiter, allowing for nested delimiter pairs.

Ruby in a Nutshell By Yukihiro Matsumoto ISBN: 0-59600-214-9 Publisher: O'Reilly
Print Publication Date: 2001/11/01
Prepared for Leanne Northrop, Safari ID: leanne.northrop@googlemail.com
User number: 724960 Copyright 2007, Safari Books Online, LLC.

`%!foo!`

`%Q!foo!`

Equivalent to double quoted string `"foo"`

`%q!foo!`

Equivalent to single quoted string `'foo'`

`%x!foo!`

Equivalent to `` `foo` `` command output

2.4.2.5. here documents

Builds strings from multiple lines. Contents span from next logical line to the line that starts with the delimiter.

```
<<FOO

FOO
```

Using quoted delimiters after <<, you can specify the quoting mechanism used for `String` literals. If a minus sign appears between << and the delimiter, you can indent the delimiter, as shown here:

```
puts <<FOO                 # String in double quotes ("")
     hello world
     FOO

     puts <<"FOO"      # String in double quotes ("")
     hello world
     FOO

     puts <<'FOO'       # String in single quotes ('')
     hello world
     FOO

     puts <<`FOO`       # String in backquotes (``)
     hello world
     FOO

     puts <<-FOO        # Delimiter can be indented
         hello world
         FOO
```

2.4.3. Symbols

A symbol is an object corresponding to an identifier or variable:

```
:foo                  # symbol for 'foo'
:$foo                 # symbol for variable '$foo'
```

2.4.4. Arrays

An array is a container class that holds a collection of objects indexed by an integer. Any kind of object may be stored in an array, and any given array can store a heterogeneous mix of object types. Arrays grow as you add elements. Arrays can be created using `array.new` or via literals. An array expression is a series of values between brackets `[ ]`:

`[]`

An empty array (with no elements)

`[1, 2, 3]`

An array of three elements

`[1, [2, 3]]`

A nested array

2.4.4.1. General delimited string array

You can construct arrays of strings using the shortcut notation, `%W`. Only whitespace characters and closing parentheses can be escaped in the following notation:

```
%w(foo bar baz)       # ["foo", "bar", "baz"]
```

2.4.5. Hashes

A hash is a collection of key-value pairs or a collection that is indexed by arbitrary types of objects.

A hash expression is a series of `key=>value` pairs between braces.

```
{key1 => val1, key2 => val2}
```

2.4.6. Regular Expressions

Regular expressions are a minilanguage used to describe patterns of strings. A regular expression literal is a pattern between slashes or between arbitrary delimiters followed by `%r`:

```
/pattern/
/pattern/im          # option can be specified
%r!/usr/local!       # general delimited regular expression
```

Regular expressions have their own power and mystery; for more on this topic, see O'Reilly's *Mastering Regular Expressions* by Jeffrey E.F. Friedl.

2.4.6.1. Regular-expression modifiers

Regular expression literals may include an optional modifier to control various aspects of matching. The modifier is specified after the second slash character, as shown previously and may be represented by one of these characters:

i

Case-insensitive

o

Substitutes only once

x

Ignores whitespace and allows comments in regular expressions

m

Matches multiple lines, recognizing newlines as normal characters

2.4.6.2. Regular-expression patterns

Except for control characters, (`+ ? . * ^ $ ( ) [ ] { } | \`), all characters match themselves. You can escape a control character by preceding it with a backslash.

Regular characters that express repetition (`* + { }`) can match very long strings, but when you follow such characters with control characters `?`, you invoke a nongreedy match that finishes at the first successful match (i.e., `+`, `*`, etc.) followed by `?` (i.e., `+?`, `*?`, etc.).

`^`

Matches beginning of line.

`$`

Matches end of line.

`.`

Matches any single character except newline. Using `m` option allows it to match newline as well.

`[...]`

Matches any single character in brackets.

`[^...]`

Matches any single character not in brackets.

`re*`

Matches 0 or more occurrences of preceding expression.

`re+`

Matches 1 or more occurrences of preceding expression.

`re?`

Matches 0 or 1 occurrence of preceding expression.

`re{ n}`

Matches exactly *`n`* number of occurrences of preceding expression.

`re{ n,}`

Matches *n* or more occurrences of preceding expression.

`re{ n, m}`

Matches at least *n* and at most *m* occurrences of preceding expression.

`a| b`

Matches either *a* or *b*.

`( re)`

Groups regular expressions and remembers matched text.

`(?imx)`

Temporarily toggles on `i`, `m`, or `x` options within a regular expression. If in parentheses, only that area is affected.

`(?-imx)`

Temporarily toggles off `i`, `m`, or `x` options within a regular expression. If in parentheses, only that area is affected.

`(?: re)`

Groups regular expressions without remembering matched text.

`(?imx: re)`

Temporarily toggles on `i`, `m`, or `x` options within parentheses.

`(?-imx: re)`

Temporarily toggles off `i`, `m`, or `x` options within parentheses.

`(?#...)`

Comment.

`(?= re)`

Specifies position using a pattern. Doesn't have a range.

`(?! re)`

Specifies position using pattern negation. Doesn't have a range.

`(?> re)`

Matches independent pattern without backtracking.

`\w`

Matches word characters.

`\W`

Matches nonword characters.

`\s`

Matches whitespace. Equivalent to [`\t\n\r\f`].

`\S`

Matches nonwhitespace.

`\d`

Matches digits. Equivalent to [0-9].

`\D`

Matches nondigits.

`\A`

Matches beginning of string.

`\Z`

Matches end of string. If a newline exists, it matches just before newline.

`\z`

Matches end of string.

`\G`

Matches point where last match finished.

`\b`

Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.

`\B`

Matches nonword boundaries.

`\n`, `\t`, etc.

Matches newlines, carriage returns, tabs, etc.

`\1...\9`

Matches *n*th grouped subexpression.

`\10...`

Matches *n*th grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

2.5. Variables

There are five types of variables in Ruby: global, instance, class, locals and constants. As you might expect, global variables are accessible globally to the program, instance variables belong to an object, class variables to a class and constants are, well... constant. Ruby uses special characters to differentiate between the different kinds of variables. At a glance, you can tell what kind of variable is being used.

Global Variables

```
$foo
```

Global variables begin with `$`. Uninitialized global variables have the value `nil` (and produce warnings with the `-w` option). Some global variables have special behavior. See Section 3.1 in Chapter 3.

Instance Variables

```
@foo
```

Instance variables begin with `@`. Uninitialized instance variables have the value `nil` (and produce warnings with the `-w` option).

Class Variables

```
@@foo
```

Class variables begin with `@@` and must be initialized before they can be used in method definitions. Referencing an uninitialized class variable produces an error. Class variables are shared among descendants of the class or module in which the class variables are defined. Overriding class variables produce warnings with the -`w` option.

Local Variables

```
foo
```

Local variables begin with a lowercase letter or _. The scope of a local variable ranges from `class`, `module`, `def`, or `do` to the corresponding `end` or from a block's opening brace to its close brace `{}`. The scope introduced by a block allows it to reference local variables outside the block, but scopes introduced by others don't. When an uninitialized local variable is referenced, it is interpreted as a call to a method that has no arguments.

Constants

```
Foo
```

Constants begin with an uppercase letter. Constants defined within a class or module can be accessed from within that class or module, and those defined outside a class or module can be accessed globally. Constants may not be defined within methods. Referencing an uninitialized constant produces an error. Making an assignment to a constant that is already initialized produces a warning, not an error. You may feel it contradicts the name "constant," but remember, this is listed under "variables."

Pseudo-Variables

In addition to the variables discussed, there are also a few *pseudo-variables*. Pseudo-variables have the appearance of local variables but behave like constants. Assignments may not be made to pseudo-variables.

`self`

The receiver object of the current method

`true`

Value representing `true`

`false`

Value representing `false`

`nil`

Value representing "undefined"; interpreted as `false` in conditionals

`__FILE__`

The name of the current source file

`__LINE__`

The current line number in the source file

Assignment

```
target = expr
```

The following elements may assign targets:

Global variables

Assignment to global variables alters global status. It isn't recommended to use (or abuse) global variables. They make programs cryptic.

Local variables

Assignment to uninitialized local variables also serves as variable declaration. The variables start to exist until the end of the current scope is reached. The lifetime of local variables is determined when Ruby parses the program.

Constants

Assignment to constants may not appear within a method body. In Ruby, re-assignment to constants isn't prohibited, but it does raise a warning.

Attributes

Attributes take the following form:

```
expr.attr
```

Assignment to attributes calls the *attr*= method of the result of *expr*.

Elements

Elements take the following form:

```
expr[arg...]
```

Assignment to elements calls the `[]`= method of the result of *expr*.

Parallel Assignment

```
target[, target...][, *target] = expr[, expr...][, *expr]
```

Targets on the left side receive assignment from their corresponding expressions on the right side. If the last left-side target is preceded by `*`, all remaining right-side values are assigned to the target as an array. If the last right-side expression is preceded by `*`, the array elements of expression are expanded in place before assignment.

If there is no corresponding expression, `nil` is assigned to the target. If there is no corresponding target, the value of right-side expression is just ignored.

Abbreviated Assignment

```
target op= expr
```

This is the abbreviated form of:

```
target = target op expr
```

The following operators can be used for abbreviated assignment:

```
+=   -=   *=   /=   %=   **=   <<=   >>=   &=   |=   ^=   &&=   ||=
```

2.6. Operators

Ruby supports a rich set of operators, as you'd expect from a modern language. However, in keeping with Ruby's object-oriented nature, most operators are in fact method calls.

This flexibility allows you to change the semantics of these operators wherever it might make sense.

2.6.1. Operator Expressions

Most operators are actually method calls. For example, `a + b` is interpreted as `a.+(b)`, where the + method in the object referred to by variable `a` is called with `b` as its argument.

For each operator (`+ - * / % ** & | ^ << >> && ||`), there is a corresponding form of abbreviated assignment operator (+= -= etc.)

Here are the operators shown in order of precedence (highest to lowest):

```
::
[]
**
+(unary) -(unary) ! ~
* / %
+ -
<< >>
&
| ^
> >= < <=
<=> == === != =~ !~
&&
||
.. ...
?:
= (and abbreviated assignment operators such as +=, -=, etc.)
not
and or
```

2.6.1.1. Nonmethod operators

The following operators aren't methods and, therefore, can't be redefined:

```
...
!
not
&&
and
||
or
```

`::`
`=`
`+=`, `-=`, (and other abbreviated assignment operators)
`? :` (ternary operator)

2.6.1.2. Range operators

Range operators function differently depending on whether or not they appear in conditionals, `if` expressions, and `while` loops.

In conditionals, they return `true` from the point right operand is `true` until left operand is `true`:

`expr1 .. expr2`

Evaluates *`expr2`* immediately after *`expr1`* turns `true`.

`expr1 ... expr2`

Evaluates *`expr2`* on the iteration after *`expr1`* turns `true`.

In other contexts, they create a range object:

`expr1 .. expr2`

Includes both expressions (*`expr1`* <= x <= *`expr2`*)

`expr1 ... expr2`

Doesn't include the last expression (*`expr1`* <= x < *`expr2`*)

2.6.1.3. Logical operators

If the value of the entire expression can be determined with the value of the left operand alone, the right operand isn't evaluated.

`&& and`

Returns `true` if both operands are `true`. If the left operand is `false`, returns the value of the left operand, otherwise returns the value of the right operand.

`|| or`

Returns `true` if either operand is `true`. If the left operand is `true`, returns the value of the left operand, otherwise returns the value of the right operand.

The operators `and` and `or` have extremely low precedence.

2.6.1.4. Ternary operator

Ternary `?:` is the conditional operator. It's another form of the `if` statement.

`a ? b : c`

If *a* is `true`, evaluates *b*, otherwise evaluates *c*. It's best to insert spaces before and after the operators to avoid mistaking the first part for the method `a?` and the second part for the symbol `:c`.

2.6.1.5. defined? operator

`defined?` is a special operator that takes the form of a method call to determine whether or not the passed expression is defined. It returns a description string of the expression, or `nil` if the expression isn't defined.

`defined? variable`

True if *variable* is initialized

```
foo = 42
defined? foo      # => "local-variable"
defined? $_       # => "global-variable"
defined? bar      # => nil (undefined)
```

`defined?` *`method_call`*

True if a method is defined (also checks arguments)

```
defined? puts      # => "method"
defined? puts(bar) # => nil (bar is not defined here)
defined? unpack    # => nil (not defined here)
```

`defined? super`

True if a method exists that can be called with `super`

```
defined? super     # => "super" (if it can be called)
defined? super     # => nil     (if it cannot be)
```

`defined? yield`

True if a code block has been passed

```
defined? yield   # => "yield" (if there is a block passed)
defined? yield   # => nil     (if there is no block)
```

2.7. Methods

Methods are the workhorses of Ruby; all of your carefully crafted algorithms live in methods on objects (and classes). In Ruby, "method" means both the named operation (e.g. "dump") and the code that a specific class provides to perform an operation.

Strictly speaking, Ruby has no functions, by which I mean code not associated with any object. (In C++, this is what you might call a "global-scope function".) All code in Ruby is a method of some object. But Ruby allows you the flexibility of having some methods appear and work just like functions in other languages, even though behind the scenes they're still just methods.

Normal Method Calls

```
obj.method([expr...[, *expr[, &expr]]])
obj.method [expr...[, *expr[, &expr]]]
obj::method([expr...[, *expr[, &expr]]])
obj::method [expr...[, *expr[, &expr]]]

method([expr...[, *expr[, &expr]]])
method [expr...[,
*expr[, &expr]]]
```

Calls a method. May take as arguments any number of `expr` followed by `*expr` and `&expr`. The last expression argument can be a hash declared directly without braces. `*expr` expands the array value of that expression and passes it to the method. `&expr` passes the `Proc` object value of that expression to the method as a block. If it isn't ambiguous, arguments need not be enclosed in parentheses. Either `.` or `::` may be used to separate the object from its method, but it is customary in Ruby code to use `::` as the separator for class methods.

Calls a method of `self`. This is the only form by which private methods may be called.

Within modules, module methods and private instance methods with the same name and definition are referred to by the general term *module functions*. This kind of method group can be called in either of the following ways:

```
Math.sin(1.0)
```

or:

```
include Math
sin(1.0)
```

You can append ! or ? to the name of a Ruby method. Traditionally, `!` is appended to a method that requires more caution than the variant of the same name without `!`. A question mark `?` is appended to a method that determines the state of a Boolean value, `true` or `false`.

Attempting to call a method without specifying either its arguments or parentheses in a context in which a local variable of the same name exists results in the method call being interpreted as a reference to the local variable, not a call to the method.

2.7.1. Specifying Blocks with Method Calls

Methods may be called with blocks of code specified that will be called from within the method.

```
method_call {[|[variable[, variable...]]|] code}
method_call do [|[variable[, variable...]]|] code end
```

Calls a method with blocks specified. The code in the block is executed after a value is passed from the method to the block and assigned to the variable (the block's argument) enclosed between `||`.

A block introduces its own scope for new local variables. The local variables that appear first in the block are local to that block. The scope introduced by a block can refer local variables of outer scope; on the other hand, the scope introduced by `class`, `module` and `def` statement can't refer outer local variables.

The form `{ . . . }` has a higher precedence than `do . . . end`. The following:

```
identifier1 identifier2 {|varizable| code}
```

actually means:

```
identifier1(identifier2 {|variable| code})
```

On the other hand:

```
identifier1 identifier2 do |variable| code end
```

actually means:

```
identifier1(identifier2) do |variable| code end
```

def Statement

```
def method([arg..., arg=default..., *arg, &arg])
code
[rescue [exception_class[, exception_class...]] [=> variable] [then]
code]...
[else
code]
[ensure
code]
end
```

Defines a method. Arguments may include the following:

arg

Mandatory argument.

arg= *default*

Optional argument. If argument isn't supplied by that which is calling the method, the *default* is assigned to *arg*. The *default* is evaluated at runtime.

* *arg*

If there are remaining actual arguments after assigning mandatory and optional arguments, they are assigned to *arg* as an array. If there is no remainder, empty array is assigned to *arg*.

& *arg*

If the method is invoked with a block, it is converted to a `Proc` object, then assigned to *arg*. Otherwise, `nil` is assigned.

Operators can also be specified as method names. For example:

```
def +(other)
  return self.value + other.value
end
```

You should specify `+@` or `-@` for a single plus or minus, respectively. As with a `begin` block, a method definition may end with `rescue`, `else`, and `ensure` clauses.

2.7.2. Singleton Methods

In Ruby, methods can be defined that are associated with specific objects only. Such methods are called singleton methods. Singleton methods are defined using `def` statements while specifying a receiver.

Defines a singleton method associated with a specific object specified by a receiver. The *receiver* may be a constant (literal) or an expression enclosed in parentheses.

def Statement for Singleton Methods

```
def
receiver.method([arg...,arg=default..., *arg, &arg])
code
[rescue [exception_class[, exception_class...]] [=> variable] [then]
    code]...
[else
    code]
[ensure
    code]
end
```

`A period .` after *receiver* can be replaced by two colons (`::`). They work the same way, but `::` is often used for class methods.

A restriction in the implementation of Ruby prevents the definition of singleton methods associated with instances of the `Fixnum` or `Symbol` class.

```
a = "foo"
def a.foo
  printf "%s(%d)\n", self, self.size
end
a.foo     # "foo" is available for a only
```

2.7.3. Method Operations

Not only can you define new methods to classes and modules, you can also make aliases to the methods and even remove them from the class.

alias Statement

```
alias new old
```

Creates an alias *new* for an existing method, operator or global variable, specified by *old*. This functionality is also available via `Module#alias_method`. When making an alias of a method, it refers the current definition of the method.

```
def foo
  puts "foo!"
  end
alias foo_orig foo
def foo
  puts "new foo!"
end
foo                 # => "new foo!"
foo_orig            # => "foo!"
```

undef Statement

```
undef method...
```

Makes method defined in the current class undefined, even if the method is defined in the superclass. This functionality is also available via `Module#undef_method`.

```
class Foo
def foo
end
end
class Bar<Foo
# Bar inherits "foo"
undef foo
end
b = Bar.new
b.foo      # error!
```

2.7.4. Other Method-Related Statements

The following statements are to be used within method definitions. The `yield` statement executes a block that is passed to the method. The `super` statement executes the overridden method of the superclass.

yield Statement

```
yield([expr...])
yield [expr...]
```

Executes the block passed to the method. The expression passed to `yield` is assigned to the block's arguments. Parallel assignment is performed when multiple expressions are passed. The output of the block, in other words the result of the last expression in the block, is returned.

super Statement

```
super
super([expr...])
superexpr...
```

`super` executes the method of the same name in the superclass. If neither arguments nor parentheses are specified, the method's arguments are passed directly to the superclass method. In other words, a call to `super( )`, which passes no arguments to the superclass method, has a different meaning from a call to `super`, where neither arguments nor parentheses are specified.

2.8. Control Structures

Ruby offers control structures that are pretty common to modern languages, but it also has a few unique ones.

if Statement

```
if conditional [then]
code
[elsif conditional [then]
code]...
[else
code]
end
```

Executes *`code`* if the *`conditional`* is `true`. True is interpreted as anything that isn't `false` or `nil`. If the *`conditional`* isn't `true`, *`code`* specified in the `else` clause is executed. An `if` expression's *`conditional`* is separated from *`code`* by the reserved word `then`, a newline, or a semicolon. The reserved word `if` can be used as a statement modifier.

```
code if conditional
```

Executes `code` if `conditional` is `true`.

unless Statement

```
unless conditional [then]
code
[else
code]
end
```

Executes code if *`conditional`* is `false`. If the *`conditional`* is `true`, *`code`* specified in the `else` clause is executed. Like `if`, `unless` can be used as a statement modifier.

```
code unless conditional
```

Executes *`code`* unless *`conditional`* is `true`.

case Statement

```
case expression
[when expression[, expression...] [then]
code]...
[else
code]
end
```

Compares the *`expression`* specified by `case` and that specified by `when` using the `===` operator and executes the *`code`* of the `when` clause that matches. The *`expression`* specified by the `when` clause is evaluated as the left operand. If no `when` clauses match, `case` executes the *`code`* of the `else` clause. A `when` statement's *`expression`* is separated from *`code`* by the reserved word `then`, a newline, or a semicolon.

while Statement

```
while conditional [do]
code
end
```

Executes *`code`* while *`conditional`* is `true`. A `while` loop's *`conditional`* is separated from *`code`* by the reserved word `do`, a newline,\, or a semicolon. The reserved word `while` can be used as statement modifier.

```
code while conditional
```

Executes*`code`* while *`conditional`* is `true`.

```
begin code end while conditional
```

If a `while` modifier follows a `begin` statement with no `rescue` or `ensure` clauses, *`code`* is executed once before *`conditional`* is evaluated.

until Statement

```
until conditional [do]
code
end

code untilconditional

begin
code
end until conditional
```

Executes `code` while `conditional` is `false`. An `until` statement's `conditional` is separated from code by the reserved word `do`, a newline, or a semicolon. Like `while`, `until` can be used as statement modifier.

Executes `code` while `conditional` is `false`.

If an `until` modifier follows a `begin` statement with no `rescue` or `ensure` clauses, `code` is executed once before `conditional` is evaluated.

for Statement

```
for variable[, variable...] in expression [do]
code
end
```

Executes `code` once for each element in `expression`. Almost exactly equivalent to:

```
expression.each do |variable[, variable...]| code end
```

except that a `for` loop doesn't create a new scope for local variables. A `for` loop's `expression` is separated from `code` by the reserved word `do`, a newline, or a semicolon.

break Statement

```
break
```

Terminates a `while`/`until` loop. Terminates a method with an associated block if called within the block (with the method returning `nil`).

next Statement

```
next
```

Jumps to the point immediately before the evaluation of a loop's conditional. Terminates execution of a block if called within a block (with `yield` or call returning `nil`).

redo Statement

```
redo
```

Jumps to the point immediately after the evaluation of the loop's conditional. Restarts `yield` or `call` if called within a block.

retry Statement

```
retry
```

Repeats a call to a method with an associated block when called from outside a `rescue` clause.

Jumps to the top of a `begin`/`end` block if called from within a `rescue` clause.

begin Statement

```
begin
code
[rescue [exception_class[, exception_class...]] [=> variable] [then]
code]...
[else
code]
[ensure
code]
end
```

The `begin` statement encloses *`code`* and performs exception handling when used together with the `rescue` and `ensure` clauses.

When a `rescue` clause is specified, exceptions belonging to the *`exception_class`* specified are caught, and the *`code`* is executed. The value of the whole `begin` enclosure is the value of its last line of `code`. If no *`exception_class`* is specified, the program is treated as if the `StandardError` class had been specified. If a *`variable`* is specified, the exception object is stored to it. The `rescue` *`exception_class`* is separated from the rest of the code by the reserved word `then`, a newline, or a semicolon. If no exceptions are raised, the `else` clause is executed if specified. If an `ensure` clause is specified, its *`code`* is always executed before the `begin`/`end` block exits, even if for some reason the block is exited before it can be completed.

rescue Statement

```
code rescue expression
```

Evaluates the *`expression`* if an exception (a subclass of `StandardError`) is raised during the execution of the *`code`*. This is exactly equivalent to:

```
begin
  code
rescue StandardError
  expression
end
```

raise method

```
raise exception_class, message
raise exception_object
raisemessage
raise
```

Raises an exception. Assumes `RuntimeError` if no *`exception_class`* is specified. Calling `raise` without arguments in a `rescue` clause re-raises the exception. Doing so outside a rescue clause raises a message-less `RuntimeError`.

BEGIN Statement

```
BEGIN {
code
}
```

Declares *`code`* to be called before the program is run.

END Statement

```
END {
code
}
```

Declares *`code`* to be called at the end of the program (when the interpreter quits).

2.9. Object-Oriented Programming

Phew, seems like a long time since I introduced Ruby as "the object-oriented scripting language," eh? But now you have everything you need to get the nitty-gritty details on how Ruby treats classes and objects. After you've mastered a few concepts and Ruby's syntax for dealing with objects, you may never want to go back to your old languages, so beware!

2.9.1. Classes and Instances

All Ruby data consists of objects that are instances of some class. Even a class itself is an object that is an instance of the `Class` class. As a general rule, new instances are created using the `new` method of a class, but there are some exceptions (such as the `Fixnum` class).

```
a = Array::new
s = String::new
o = Object::new
```

class Statement

```
class class_name [< superclass]
code
end
```

Defines a class. A *`class_name`* must be a constant. The defined class is assigned to that constant. If a class of the same name already exists, the class and *`superclass`* must match, or the *`superclass`* must not be specified, in order for the features of the new class definition to be added to the existing class. `class` statements introduce a new scope for local variables.

2.9.2. Methods

Class methods are defined with the `def` statement. The `def` statement adds a method to the innermost class or module definition surrounding the `def` statement. A `def` statement outside a class or module definition (at the top level) adds a method to the `Object` class itself, thus defining a method that can be referenced anywhere in the program.

When a method is called, Ruby searches for it in a number of places in the following order:

1. Among the methods defined in that object (i.e., singleton methods).
2. Among the methods defined by that object's class.
3. Among the methods of the modules included by that class.
4. Among the methods of the superclass.
5. Among the methods of the modules included by that superclass.
6. Repeats Steps 4 and 5 until the top-level object is reached.

2.9.3. Singleton Classes

Attribute definitions for a specific object can be made using the class definition construction. Uses for this form of class definition include the definition and a collection of singleton methods.

```
class << object
   code
end
```

Creates a virtual class for a specific object, defining the properties (methods and constants) of the class using the class definition construction.

2.9.4. Modules

A module is similar to a class except that it has no superclass and can't be instantiated. The `Module` class is the superclass of the `Class` class.

module Statement

```
module module_name
  code
end
```

A `module` statement defines a module. *`module_name`* must be a constant. The defined module is assigned to that constant. If a module of the same name already exists, the features of the new module definition are added to the existing module. `module` statements introduce a new scope for local variables.

2.9.5. Mix-ins

Properties (methods and constants) defined by a module can be added to a class or another module with the `include` method. They can also be added to a specific object using the extend method. See `Module#include` in Section 3.4.9, and the `Object#extend` in Section 3.4.1.

2.9.6. Method Visibility

There are three types of method visibility:

`Public`

Callable from anywhere

`Protected`

Callable only from instances of the same class

`Private`

Callable only in functional form (i.e., without the receiver specified)

Method visibility is defined using the `public`, `private`, and `protected` methods in classes and modules.

`public(` *`[symbol ...])`*

Makes the method specified by `symbol` public. The method must have been previously defined. If no arguments are specified, the visibility of all subsequently defined methods in the class or module is made public.

`protected([` *`symbol`*`...])`

Makes the method specified by `symbol` protected. The method must have been previously defined. If no arguments are specified, the visibility of all subsequently defined methods in the class or module is made protected.

`private([` *`symbol`*`...])`

Makes the method specified by `symbol` private. The method must have been previously defined. If no arguments are specified, the visibility of all subsequently defined methods in the class or module is made private.

2.9.7. Object Initialization

Objects are created using the `new` method of each object's class. After a `new` object is created by the `new` method, the object's `initialize` method is called with the arguments of the `new` method passed to it. Blocks associated with the `new` method are also passed directly to `initialize`. For consistency, you should initialize objects by redefining the `initialize` method, rather than the `new` method. The visibility of methods named `initialize` is automatically made private.

2.9.8. Attributes

Attributes are methods that can be referenced and assigned to externally as if they were variables. For example, the `Process` module attribute `egid` can be manipulated in the following way:

```
Process.egid      # Reference
Process.egid=id   # Assignment
```

These are actually two methods, one that takes no argument and another with a name ending with = that takes one argument. Methods that form such attributes are referred to as *accessor* methods.

2.9.9. Hooks

Ruby notifies you when a certain event happens, as shown in Table 2-2.

Table 2-2. Events and their hook methods

Event	Hook method	Of
Defining an instance method	`method_added`	Class
Defining a singleton method	`singleton_method_added`	Object
Make subclass	`inherited`	Superclass

These methods are called *hooks*. Ruby calls hook methods when the specific event occurs (at runtime). The default behavior of these methods is to do nothing. You have to override the method if you want to do something on a certain event:

```
class Foo
  def Foo::inherited(sub)
    printf "you made subclass of Foo, named %s\n", sub.name
  end
end
class Bar<Foo  # prints "you made subclass of Foo, named Bar"
end
```

There are other types of hook methods used by the mix-in feature. They are called by `include` and `extend` to do the actual mixing-in, as shown in Table 2-3. You can use these as hooks, but you have to call `super` when you override them.

Table 2-3. Mix-In hook methods

Event	Hook method	Of	From
Mixing in a module	`append_features`	Mix-in module	`Module#include`
Extending a object	`extend_object`	Mix-in module	`Object#extend`

Ruby 1.7 and later provide more hooks. See Chapter 6 for more information on future versions.

2.10. Security

Ruby is portable and can easily use code distributed across a network. This property gives you tremendous power and flexibility but introduces a commensurate burden: how do you use this capability without possibly causing damage?

Part of the answer lies in Ruby's security system, which allows you to "lock down" the Ruby environment when executing code that may be suspect. Ruby calls such data and code *tainted*. This feature introduces mechanisms that allow you to decide how and when potentially "dangerous" data or code can be used inside your Ruby scripts.

2.10.1. Restricted Execution

Ruby can execute programs with *security checking* turned on. The global variable `$SAFE` determines the level of the security check. The default safe level is 0, unless specified explicitly by the command-line option `-T`, or the Ruby script is run `setuid` or `setgid`.

`$SAFE` can be altered by assignment, but it isn't possible to lower the value of it:

```
$SAFE=1                # upgrade the safe level
$SAFE=4                #  upgrade the safe level even higher
$SAFE=0                # SecurityError!  you can't do it
```

`$SAFE` is thread local; in other words, the value of `$SAFE` in a thread may be changed without affecting the value in other threads. Using this feature, threads can be sandboxed for untrusted programs.

```
Thread::start {        # starting "sandbox" thread
  $SAFE = 4            # for this thread only
  ...                  # untrusted code
}
```

Level 0

Level 0 is the default safe level. No checks are performed on tainted data.

Any externally supplied string from `IO`, environment variables, and `ARGV` is automatically flagged as tainted.

The environment variable `PATH` is an exception. Its value is checked, and tainted only if any directory in it is writable by everybody.

Level 1

In this level, potentially dangerous operations using tainted data are forbidden. This is a suitable level for programs that handle untrusted input, such as CGI.

- Environment variables `RUBYLIB` and `RUBYOPT` are ignored at startup.
- Current directory (.) isn't included in `$LOAD_PATH`.
- The command-line options `-e`, `-i`, `-I`, `-r`, `-s`, `-S`, and `-X` are prohibited.
- Process termination if the environment variable `PATH` is tainted.
- Invoking methods and class methods of `Dir`, `IO`, `File`, and `FileTest` for tainted arguments is prohibited.
- Invoking `test`, `eval`, `require`, `load`, and `trap` methods for tainted argument is prohibited.

Level 2

In this level, potentially dangerous operations on processes and files are forbidden, in addition to all restrictions in level 1. The following operations are prohibited:

```
Dir::chdir
Dir::chroot
Dir::mkdir
Dir::rmdir
```

`File::chown`
`File::chmod`
`File::umask`
`File::truncate`
`File#lstat`
`File#chmod`
`File#chown`
`File#truncate`
`File#flock`
`IO#ioctl`
`IO#fctrl`
Methods defined in the `FileTest` module
`Process::fork`
`Process::setpgid`
`Process::setsid`
`Process::setpriority`
`Process::egid=`
`Process::kill`
`load` from a world-writable directory
`syscall`
`exit!`
`trap`

Level 3

In this level, all newly created objects are considered tainted, in addition to all restrictions in Level 2.

- All objects are created tainted.
- `Object#untaint` is prohibited.
- `Proc` objects retain current safe level to restore when their `call` methods are invoked.

Level 4

In this level, modification of global data is forbidden, in addition to all restrictions in Level 3. `eval` is allowed again in this level, since all dangerous operations are blocked in this level.

```
def safe_eval(str)
Thread::start {           # start sandbox thread
  $SAFE = 4               # upgrade safe level
  eval(str)               # eval in the sandbox
}.value                   # retrieve result
end

eval('1 + 1')             # => 2
eval('system "rm -rf /"') # SecurityError
```

The following operations are prohibited:

- `Object#taint`
- `autoload`, `load`, and `include`
- Modifying `Object` class
- Modifying untainted objects
- Modifying untainted classes or modules
- Retrieving meta information (e.g., variable list)
- Manipulating instance variables
- Manipulating threads other than current
- Accessing thread local data
- Terminating process (by `exit`, `abort`)
- File input/output
- Modifying environment variables
- `srand`

Ruby in a Nutshell By Yukihiro Matsumoto ISBN: 0-59600-214-9 Publisher: O'Reilly
Print Publication Date: 2001/11/01