Part III

Ruby Crystallized

Chapter 22

The Ruby Language

This chapter is a bottom-up look at the Ruby language. Most of what appears here is the syntax and semantics of the language itself—we mostly ignore the built-in classes and modules (these are covered in depth starting on page 442). However, Ruby sometimes implements features in its libraries that in most languages would be part of the basic syntax. We've included these methods here and have tried to flag them with "Library" in the margin.

The contents of this chapter may look familiar—with good reason. We've covered just about all of this in the earlier tutorial chapters. Consider this chapter to be a self-contained reference to the core Ruby language.

1.9 Source File Encoding

Ruby programs are by default written in 7-bit ASCII, also called US-ASCII. If a code set other than 7-bit ASCII is to be used, place a comment containing coding: followed by the name of an encoding on its own on the first line of each source file containing non-ASCII characters. The coding: comment can be on the second line of the file if the first line is a shebang comment. Ruby skips characters in the comment before the word coding:

`# coding: utf-8`	`# -*- encoding: iso-8859-1 -*-`	`#!/usr/bin/ruby`
		`# fileencoding: us-ascii`
UTF-8 source...	*ISO-8859-1 source...*	*ASCII source...*

Source Layout

Ruby is a line-oriented language. Ruby expressions and statements are terminated at the end of a line unless the parser can determine that the statement is incomplete—for example, if the last token on a line is an operator or comma. A semicolon can be used to separate multiple expressions on a line. You can also put a backslash at the end of a line to continue it onto the next. Comments start with # and run to the end of the physical line. Comments are ignored during syntax analysis.

```
a = 1
b = 2; c = 3
d = 4 + 5 +
    6 + 7        # no '\' needed
e = 8 + 9   \
    + 10         # '\' needed
```

Physical lines between a line starting with =begin and a line starting with =end are ignored by Ruby and may be used to comment out sections of code or to embed documentation.

Ruby reads its program input in a single pass, so you can pipe programs to the Ruby interpreter's standard input stream:

```
echo 'puts "Hello"' | ruby
```

If Ruby comes across a line anywhere in the source containing just "__END__", with no leading or trailing whitespace, it treats that line as the end of the program—any subsequent lines will not be treated as program code. However, these lines can be read into the running program using the global IO object DATA, described on page 343.

BEGIN and END Blocks

Every Ruby source file can declare blocks of code to be run as the file is being loaded (the BEGIN blocks) and after the program has finished executing (the END blocks):

```
BEGIN {
  begin code
}

END {
  end code
}
```

A program may include multiple BEGIN and END blocks. BEGIN blocks are executed in the order they are encountered. END blocks are executed in reverse order.

General Delimited Input

As well as the normal quoting mechanism, alternative forms of literal strings, arrays, regular expressions, and shell commands are specified using a generalized delimited syntax. All these literals start with a percent character, followed by a single character that identifies the literal's type. These characters are summarized in Table 22.1 on the next page; the actual literals are described in the corresponding sections later in this chapter.

Following the type character is a delimiter, which can be any nonalphabetic or nonmultibyte character. If the delimiter is one of the characters (, [, {, or <, the literal consists of the characters up to the matching closing delimiter, taking account of nested delimiter pairs. For all other delimiters, the literal comprises the characters up to the next occurrence of the delimiter character.

Table 22.1. General Delimited Input

Type	Meaning	See Page
%q	Single-quoted string	328
%Q, %	Double-quoted string	328
%w, %W	Array of strings	330
%r	Regular expression pattern	332
%s	A symbol	331
%x	Shell command	344

```
%q/this is a string/
%q-string-
%q(a (nested) string)
```

Delimited strings may continue over multiple lines; the line endings and all spaces at the start of continuation lines will be included in the string:

```
meth = %q{def fred(a)
            a.each {|i| puts i }
          end}
```

The Basic Types

The basic types in Ruby are numbers, strings, arrays, hashes, ranges, symbols, and regular expressions.

Integer and Floating-Point Numbers

Ruby integers are objects of class Fixnum or Bignum. `Fixnum` objects hold integers that fit within the native machine word minus 1 bit. Whenever a Fixnum exceeds this range, it is automatically converted to a `Bignum` object, whose range is effectively limited only by available memory. If an operation with a `Bignum` result has a final value that will fit in a `Fixnum`, the result will be returned as a `Fixnum`.

Integers are written using an optional leading sign and an optional base indicator (`0` or `0o` for octal, `0d` for decimal, `0x` for hex, or `0b` for binary), followed by a string of digits in the appropriate base. Underscore characters are ignored in the digit string.

```
123456                  => 123456   # Fixnum
0d123456                => 123456   # Fixnum
123_456                 => 123456   # Fixnum - underscore ignored
-543                    => -543     # Fixnum - negative number
0xaabb                  => 43707    # Fixnum - hexadecimal
0377                    => 255      # Fixnum - octal
0o377                   => 255      # Fixnum - octal
-0b10_1010              => -42      # Fixnum - binary (negated)
123_456_789_123_456_789 => 123456789123456789 # Bignum
```

A numeric literal with a decimal point and/or an exponent is turned into a Float object, corresponding to the native architecture's double data type. You must follow the decimal point with a digit; if you write 1.e3, Ruby tries to invoke the method e3 on the Fixnum 1. You must place at least one digit before the decimal point.

```
12.34       # =>  12.34
-0.1234e2   # =>  -12.34
1234e-2     # =>  12.34
```

Rational and Complex Numbers

1.9

Classes that support rational numbers (ratios of integers) and complex numbers are built into the Ruby interpreter. However, Ruby provides no language-level support for these numeric types. There are for rational or complex literals, for example. See the descriptions of Complex and Rational on pages 473 and 660 for more information.

Strings

Ruby provides a number of mechanisms for creating literal strings. Each generates objects of type String. The different mechanisms vary in terms of how a string is delimited and how much substitution is done on the literal's content. Literal strings are encoded using the source encoding of the file that contains them.

Single-quoted string literals ('*stuff*' and %q/*stuff*/) undergo the least substitution. Both convert the sequence \\ into a single backslash, and the form with single quotes converts \' into a single quote. All other backslashes appear literally in the string.

```
'hello'                      # =>  hello
'a backslash \'\\\''         # =>  a backslash '\'
%q/simple string/            # =>  simple string
%q(nesting (really) works)   # =>  nesting (really) works
%q no_blanks_here ;          # =>  no_blanks_here
```

Double-quoted strings ("*stuff*", %Q/*stuff*/, and %/*stuff*/) undergo additional substitutions, shown in Table 22.2 on the following page.

```
a  = 123
"\123mile"                      # =>  Smile
"Greek pi: \u03c0"              # =>  Greek pi: π
"Greek \u{70 69 3a 20 3c0}"     # =>  Greek pi: π
"Say \"Hello\""                 # =>  Say "Hello"
%Q!"I said 'nuts'," I said!     # =>  "I said 'nuts'," I said
%Q{Try #{a + 1}, not #{a - 1}}  # =>  Try 124, not 122
%<Try #{a + 1}, not #{a - 1}>   # =>  Try 124, not 122
"Try #{a + 1}, not #{a - 1}"    # =>  Try 124, not 122
%{ #{ a = 1; b = 2; a + b } }   # =>   3
```

1.9

Last, and probably least (in terms of usage), you can get the string corresponding to an ASCII character by preceding that character with a question mark. You can use the backslash escapes shown in Table 22.2 on the next page.

Table 22.2. Substitutions in Double-Quoted Strings

\a	Bell/alert (0x07)	\nnn	Octal *nnn*
\b	Backspace (0x08)	\xnn	Hex *nn*
\e	Escape (0x1b)	\c*x*	Control-*x*
\f	Formfeed (0x0c)	\C-*x*	Control-*x*
\n	Newline (0x0a)	\M-*x*	Meta-*x*
\r	Return (0x0d)	\M-\C-*x*	Meta-control-*x*
\s	Space (0x20)	*x*	*x*
\t	Tab (0x09)	#{code}	Value of *code*
\v	Vertical tab (0x0b)	\uxxxx	Unicode character
		\u{xx xx xx}	Unicode characters

```
?a          # => "a"      (ASCII character)
?\n         # => "\n"     (newline (0x0a))
?\C-a       # => "\x01"   (control a = 0x65 & 0x9f = 0x01)
?\M-a       # => "\xE1"   (meta sets bit 7)
?\M-\C-a    # => "\x81"   (meta and control a)
?\C-?       # => "\x7F"   (delete character)
```

Strings can continue across multiple input lines, in which case they will contain newline characters. It is also possible to use here documents to express long string literals. Whenever Ruby parses the sequence <<*identifier* or <<*quoted string*, it replaces it with a string literal built from successive logical input lines. It stops building the string when it finds a line that starts with *identifier* or *quoted string*. You can put a minus sign immediately after the << characters, in which case the terminator can be indented from the left margin. If a quoted string was used to specify the terminator, its quoting rules will be applied to the here document; otherwise, double-quoting rules apply.

```
print <<HERE
Double quoted \
here document.
It is #{Time.now}
HERE
print <<-'THERE'
    This is single quoted.
    The above used #{Time.now}
    THERE
```

produces:

```
Double quoted here document.
It is 2009-04-13 13:26:11 -0500
    This is single quoted.
    The above used #{Time.now}
```

Adjacent single- and double-quoted strings in the input are concatenated to form a single `String` object:

```
'Con' "cat" 'en' "ate"   # =>   "Concatenate"
```

Every time a string literal is used in an assignment or as a parameter, a new String object is created:

```
3.times do
  print 'hello'.object_id, " "
end
```

produces:

```
338430 338370 338330
```

The documentation for class String starts on page 670.

Ranges

Outside the context of a conditional expression, *expr..expr* and *expr...expr* construct Range objects. The two-dot form is an inclusive range; the one with three dots is a range that excludes its last element. See the description of class Range on page 656 for details. Also see the description of conditional expressions on page 348 for other uses of ranges.

Arrays

Literals of class Array are created by placing a comma-separated series of object references between square brackets. A trailing comma is ignored.

```
arr = [ fred, 10, 3.14, "This is a string", barney("pebbles"), ]
```

Arrays of strings can be constructed using the shortcut notations %w and %W. The lowercase form extracts space-separated tokens into successive elements of the array. No substitution is performed on the individual strings. The uppercase version also converts the words to an array but performs all the normal double-quoted string substitutions on each individual word. A space between words can be escaped with a backslash. This is a form of general delimited input, described on pages 326–327.

```
arr = %w( fred wilma barney betty great\ gazoo )
arr   # =>   ["fred", "wilma", "barney", "betty", "great gazoo"]
arr = %w( Hey!\tIt is now -#{Time.now}- )
arr   # =>   ["Hey!\tIt", "is", "now", "-#{Time.now}-"]
arr = %W( Hey!\tIt is now -#{Time.now}- )
arr   # =>   ["Hey! It", "is", "now", "-2009-04-13 13:26:11 -0500-"]
```

Hashes

A literal Ruby Hash is created by placing a list of key/value pairs between braces. Keys and values can be separated by the sequence =>.[1]

```
colors = { "red"   => 0xf00, "green" => 0x0f0, "blue"  => 0x00f }
```

1. As of Ruby 1.9, a comma may no longer be used to separate keys and values in hash literals. A comma still appears between each key/value pair.

If the keys are symbols, you can use this alternative notation:

```
colors = { red: 0xf00, green: 0x0f0, blue: 0x00f }
```

The keys and/or values in a particular hash need not have the same type.

Requirements for a Hash Key

Hash keys must respond to the message hash by returning a hash code, and the hash code for a given key must not change. The keys used in hashes must also be comparable using eql?. If eql? returns true for two keys, then those keys must also have the same hash code. This means that certain classes (such as Array and Hash) can't conveniently be used as keys, because their hash values can change based on their contents.

If you keep an external reference to an object that is used as a key and use that reference to alter the object, thus changing its hash code, the hash lookup based on that key may not work. You can force the hash to be reindexed by calling its rehash method.

```
arr = [1, 2, 3]
hash = { arr => 'value' }
hash[arr]   # =>   "value"
arr[1] = 99
hash        # =>   {[1, 99, 3]=>"value"}
hash[arr]   # =>   nil
hash.rehash
hash[arr]   # =>   "value"
```

Because strings are the most frequently used keys and because string contents are often changed, Ruby treats string keys specially. If you use a String object as a hash key, the hash will duplicate the string internally and will use that copy as its key. The copy will be frozen. Any changes made to the original string will not affect the hash.

If you write your own classes and use instances of them as hash keys, you need to make sure that either (a) the hashes of the key objects don't change once the objects have been created or (b) you remember to call the Hash#rehash method to reindex the hash whenever a key hash *is* changed.

Symbols

A Ruby symbol is an identifier corresponding to a string of characters, often a name. You construct the symbol for a name by preceding the name with a colon, and you can construct the symbol for an arbitrary string by preceding a string literal with a colon. Substitution occurs in double-quoted strings. A particular name or string will always generate the same symbol, regardless of how that name is used within the program. You can also use the %s delimited notation to create a symbol.

```
:Object
:my_variable
:"Ruby rules"
a = "cat"
:'catsup'                  # =>   :catsup
:"#{a}sup"                 # =>   :catsup
:'#{a}sup'                 # =>   :"\#{a}sup"
%s{symbol}                 # =>   :symbol
%s{ symbol with spaces }   # =>   :" symbol with spaces "
```

Other languages call this process *interning* and call symbols *atoms*.

Regular Expressions

This section contains a summary on the Oniguruma regular expression engine used by Ruby. See Chapter 7 on page 117 for a detailed description of regular expressions.

Regular expression literals are objects of type Regexp. They are created explicitly by calling Regexp.new or implicitly by using the literal forms, */pattern/* and %r{*pattern*}. The %r construct is a form of general delimited input (described on pages 326–327).

```
/pattern/
/pattern/options
%r{pattern}
%r{pattern}options
Regexp.new( 'pattern' [ , options ] )
```

options is one or more of i (case insensitive), o (substitute once), m (. matches newline), and x (allow spaces and comments). You can additionally override the default encoding of the pattern with n (no encoding-ASCII), e (EUC), s (Shift_JIS), or u (UTF-8).

Regular Expression Patterns

1.9 (This section contains many differences from previous versions of this book. Ruby 1.9 uses the Oniguruma regular expression engine.)[2]

characters — All except ., |, (,), [, \, ^, {, +, $, *, and ? match themselves. To match one of these characters, precede it with a backslash.

\a \c*x* \e \f \r \t \u*nnnn* \v \x*nn* *nnn* \C-\M-*x* \C-*x* \M-*x*
Match the character derived according to Table 22.2 on page 329.

^, $ — Match the beginning/end of a line.

\A, \z, \Z — Match the beginning/end of the string. \Z ignores trailing \n.

\d, \h — Match any decimal digit (or Unicode *Decimal_Number*), hexadecimal digit ([0-9a-fA-F]).

\s — Matches any whitespace character: tab, newline, vertical tab, form feed, return, and space. For Unicode, add *Line_Separator* codepoints.

2. Some of the information here is based on http://www.geocities.jp/kosako3/oniguruma/doc/RE.txt.

\w	Matches any word character: alphanumerics and underscores. For Unicode, add in the codepoints in *Connector_Punctuation*, *Letter*, *Mark*, and *Number*.
\D,\H, \S, \W	The negated forms of \d, \h, \s, and \w, matching characters that are not digits, hexadecimal digits, whitespace, or word characters.
\b, \B	Match word/nonword boundaries.
\G	The position where a previous repetitive search completed.
\p{*property*}, \P{*property*}, \p{!*property*}	Match a character that is in/not in the given property (see Table 7.3 on page 126).
. (period)	Appearing outside brackets, matches any character except a newline. (With the /m option, it matches newline, too).
[*characters*]	Matches a single character from the specified set. See page 123.
*re**	Matches zero or more occurrences of *re*.
re+	Matches one or more occurrences of *re*.
re{m,n}	Matches at least "m" and at most "n" occurrences of *re*.
re{m,}	Matches at least "m" occurrences of *re*.
re{,n}	Matches at most "n" occurrences of *re*.
re{m}	Matches exactly "m" occurrences of *re*.
re?	Matches zero or one occurrence of *re*.
	The ?, *, +, and {m,n} modifiers are greedy by default. Append a question mark to make them minimal, and append a plus sign to make them possessive (that is, they are greedy and will not backtrack).
re1|*re2*	Matches either *re1* or *re2*.
(...)	Group regular expressions and introduce extensions.
#{...}	Substitutes expression in the pattern, as with strings. By default, the substitution is performed each time a regular expression literal is evaluated. With the /o option, it is performed just the first time.
\0, \1, \2, ... *n*, \&, \`, \', \+	Substitute the value matched by the *n*th grouped subexpression or by the entire match, pre- or postmatch, or the highest group.
(?# *comment*)	Inserts a comment into the pattern.
(?:*re*)	Makes *re* into a group without generating backreferences.
(?=*re*), (?!*re*)	Matches if *re* is/is not at this point but does not consume it.
(?<=*re*), (?<!*re*)	Matches if *re* is/is not before this point but does not consume it.
(?>*re*)	Matches *re*, but inhibits subsequent backtracking.
(?imx), (?-imx)	Turn on/off the corresponding i, m, or x option. If used inside a group, the effect is limited to that group.
(?imx:*re*), (?-imx:*re*)	Turn on/off the i, m, or x option for *re*.

\n, \k'n', and \k<n>
: The n^{th} captured subpattern.

(?<name>...) or (?'name'...)
: Name the string captured by the group.

\k<name> or \k'name'
: The contents of the named group.

\k<name>+n/1 or \k'name'+/-n
: The contents of the named group at the given relative nesting level.

\g<name> or \g<number>
: Invoke the named or numbered group.

Names

Ruby names are used to refer to constants, variables, methods, classes, and modules. The first character of a name helps Ruby to distinguish its intended use. Certain names, listed in Table 22.3 on the next page, are reserved words and should not be used as variable, method, class, or module names.

Method names are described in the section beginning on page 351.

In these descriptions, *Uppercase letter* means *A* though *Z*, and *digit* means *0* through *9*. *lowercase letter* means the characters *a* though *z*, as well as _ , the underscore. In addition, any non-7-bit characters that are valid in the current encoding are considered to be lowercase.[3]

A *name* is an uppercase letter, a lowercase letter, or an underscore, followed by *name characters*: any combination of upper- and lowercase letters, underscores, and digits.

A **local variable name** consists of a lowercase letter followed by name characters. It is conventional to use underscores rather than camelCase to write multiword names, but the interpreter does not enforce this:

```
fred  anObject  _x  three_two_one
```

If the source file encoding is UTF-8, δelta and été are both valid local variable names.

An **instance variable name** starts with an "at" sign (@) followed by a name. It is generally a good idea to use a lowercase letter after the @.

```
@name  @_  @size
```

A **class variable name** starts with two "at" signs (@@) followed by a name.

```
@@name  @@_  @@Size
```

A **constant name** starts with an uppercase letter followed by name characters. Class names and module names are constants and follow the constant naming conventions.

3. Such names will not be usable from other source files with different encoding.

Table 22.3. Reserved Words

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

By convention, constant object references are normally spelled using uppercase letters and underscores throughout, while class and module names are MixedCase:

```
module Math
  ALMOST_PI = 22.0/7.0
end
class BigBlob
end
```

Global variables, and some special system variables, start with a dollar sign ($) followed by name characters. In addition, Ruby defines a set of two-character global variable names in which the second character is a punctuation character. These predefined variables are listed starting on page 339. Finally, a global variable name can be formed using $- followed by a single letter or underscore. These latter variables typically mirror the setting of the corresponding command-line option (see the table starting on page 341 for details):

```
$params  $PROGRAM  $!  $_  $-a  $-K
```

Variable/Method Ambiguity

When Ruby sees a name such as a in an expression, it needs to determine whether it is a local variable reference or a call to a method with no parameters. To decide which is the case, Ruby uses a heuristic. As Ruby parses a source file, it keeps track of symbols that have been assigned to. It assumes that these symbols are variables. When it subsequently comes across a symbol that could be a variable or a method call, it checks to see whether it has seen a prior assignment to that symbol. If so, it treats the symbol as a variable; otherwise, it treats it as a method call. As a somewhat pathological case of this, consider the following code fragment, submitted by Clemens Hintze:

```
def a
  print "Function 'a' called\n"
  99
end
for i in 1..2
  if i == 2
    print "a=", a, "\n"
  else
    a = 1
    print "a=", a, "\n"
  end
end
```

produces:

```
a=1
Function 'a' called
a=99
```

During the parse, Ruby sees the use of a in the first print statement and, because it hasn't yet seen any assignment to a, assumes that it is a method call. By the time it gets to the second print statement, though, it *has* seen an assignment and so treats a as a variable.

Note that the assignment does not have to be executed—Ruby just has to have seen it. This program does not raise an error.

```
a = 1 if false; a
```

Variables and Constants

Ruby variables and constants hold references to objects. Variables themselves do not have an intrinsic type. Instead, the type of a variable is defined solely by the messages to which the object referenced by the variable responds.[4]

A Ruby *constant* is also a reference to an object. Constants are created when they are first assigned to (normally in a class or module definition). Ruby, unlike less flexible languages, lets you alter the value of a constant, although this will generate a warning message:

```
MY_CONST = 1
MY_CONST = 2   # generates a warning
```

produces:

```
/tmp/prog.rb:2: warning: already initialized constant MY_CONST
```

Note that although constants should not be changed, you can alter the internal states of the objects they reference:[5]

```
MY_CONST = "Tim"
MY_CONST[0] = "J"   # alter string referenced by constant
MY_CONST   # =>   "Jim"
```

Assignment potentially *aliases* objects, creating two references to the same object.

Scope of Constants and Variables

Constants defined within a class or module may be accessed unadorned anywhere within the class or module. Outside the class or module, they may be accessed using the scope operator, :: prefixed by an expression that returns the appropriate class or module object. Constants defined outside any class or module may be accessed unadorned or by using the

4. When we say that a variable is not typed, we mean that any given variable can at different times hold references to objects of many different types.

5. You can freeze objects to prevent this.

scope operator :: with no prefix. Constants may not be defined in methods. Constants may be added to existing classes and modules from the outside by using the class or module name and the scope operator before the constant name.

```
OUTER_CONST = 99
class Const
  def get_const
    CONST
  end
  CONST = OUTER_CONST + 1
end
Const.new.get_const   # =>   100
Const::CONST          # =>   100
::OUTER_CONST         # =>   99
Const::NEW_CONST = 123
```

Global variables are available throughout a program. Every reference to a particular global name returns the same object. Referencing an uninitialized global variable returns nil.

Class variables are available throughout a class or module body. Class variables must be initialized before use. A class variable is shared among all instances of a class and is available within the class itself.

```
class Song
  @@count = 0
  def initialize
    @@count += 1
  end
  def Song.get_count
    @@count
  end
end
```

Class variables belong to the innermost enclosing class or module. Class variables used at the top level are defined in Object and behave like global variables. Class variables defined within singleton methods belong to the top level (although this usage is deprecated and generates a warning). In Ruby 1.9, class variables are private to the defining class:

1.9

```
class Holder
  @@var = 99
  def Holder.var=(val)
    @@var = val
  end
  def var
    @@var
  end
end
@@var = "top level variable"
a = Holder.new
a.var   # =>   "top level variable"
```

```
Holder.var = 123
a.var   # =>   123

# This references the top-level object
def a.get_var
  @@var
end
a.get_var   # =>   "top level variable"
```

Class variables are inherited by children but are unique across children:

```
class Top
  @@A = 1
  @@B = 1
  def dump
    puts values
  end
  def values
    "#{self.class.name}: @@A = #@@A, @@B = #@@B"
  end
end
class MiddleOne < Top
  @@B = 2
  @@C = 2
  def values
    super + ", C = #@@C"
  end
end
class MiddleTwo < Top
  @@B = 3
  @@C = 3
  def values
    super + ", C = #@@C"
  end
end
class BottomOne < MiddleOne; end
class BottomTwo < MiddleTwo; end

Top.new.dump
MiddleOne.new.dump
MiddleTwo.new.dump
BottomOne.new.dump
BottomTwo.new.dump
```

produces:

```
Top: @@A = 1, @@B = 3
MiddleOne: @@A = 1, @@B = 3, C = 2
MiddleTwo: @@A = 1, @@B = 3, C = 3
BottomOne: @@A = 1, @@B = 3, C = 2
BottomTwo: @@A = 1, @@B = 3, C = 3
```

I recommend against using class variables for this reason.

Instance variables are available within instance methods throughout a class body. Referencing an uninitialized instance variable returns nil. Each instance of a class has a unique set of instance variables. Instance variables are not available to class methods (although classes [and modules] also may have instance variables—see page 387).

Local variables are unique in that their scopes are statically determined but their existence is established dynamically.

A local variable is created dynamically when it is first assigned a value during program execution. However, the scope of a local variable is statically determined to be the immediately enclosing block, method definition, class definition, module definition, or top-level program. Referencing a local variable that is in scope but that has not yet been created generates a NameError exception. Local variables with the same name are different variables if they appear in disjoint scopes.

Method parameters are considered to be variables local to that method.

Block parameters are assigned values when the block is invoked.

If a local variable is first assigned in a block, it is local to the block.

If a block uses a variable that is previously defined in the scope containing the block's definition, then the block will share that variable with the scope. There are two exceptions to this. Block parameters are always local to the block. In addition, variables listed after a semicolon at the end of the block parameter list are also always local to the block.

```
a = 1
b = 2
c = 3
some_method { |b; c| a = b + 1; c = a + 1; d = c + 1 }
```

In this previous example, the variable a inside the block is shared with the surrounding scope. The variables b and c are not shared, because they are listed in the block's parameter list, and the variable d is not shared because it occurs only inside the block.

A block takes on the set of local variables in existence at the time that it is created. This forms part of its binding. Note that although the binding of the variables is fixed at this point, the block will have access to the *current* values of these variables when it executes. The binding preserves these variables even if the original enclosing scope is destroyed.

The bodies of while, until, and for loops are part of the scope that contains them; previously existing locals can be used in the loop, and any new locals created will be available outside the bodies afterward.

Predefined Variables

The following variables are predefined in the Ruby interpreter. In these descriptions, the notation [r/o] indicates that the variables are read-only; an error will be raised if a program attempts to modify a read-only variable. After all, you probably don't want to change the meaning of true halfway through your program (except perhaps if you're a politician). Entries marked [thread] are thread local.

Many global variables look something like Snoopy swearing: $_, $!, $&, and so on. This is for "historical" reasons because most of these variable names come from Perl. If you find memorizing all this punctuation difficult, you may want to take a look at the library file called English, documented on page 748, which gives the commonly used global variables more descriptive names.

In the tables of variables and constants that follow, we show the variable name, the type of the referenced object, and a description.

Exception Information

$!	Exception	The exception object passed to raise. [thread]
$@	Array	The stack backtrace generated by the last exception. See Kernel#caller on page 567 for details. [thread]

Pattern Matching Variables

These variables (except $=) are set to nil after an unsuccessful pattern match.

$&	String	The string matched (following a successful pattern match). This variable is local to the current scope. [r/o, thread]
$+	String	The contents of the highest-numbered group matched following a successful pattern match. Thus, in "cat" =~/(c|a)(t|z)/, $+ will be set to "t." This variable is local to the current scope. [r/o, thread]
$`	String	The string preceding the match in a successful pattern match. This variable is local to the current scope. [r/o, thread]
$'	String	The string following the match in a successful pattern match. This variable is local to the current scope. [r/o, thread]
$1...$n	String	The contents of successive groups matched in a successful pattern match. In "cat" =~/(c|a)(t|z)/, $1 will be set to "a" and $2 to "t." This variable is local to the current scope. [r/o, thread]
$~	MatchData	An object that encapsulates the results of a successful pattern match. The variables $&, $`, $', and $1 to $9 are all derived from $~. Assigning to $~ changes the values of these derived variables. This variable is local to the current scope. [thread]

1.9 The variable $=, which previously controlled case-insensitive matches, has been removed from Ruby 1.9.

Input/Output Variables

$/	String	The input record separator (newline by default). This is the value that routines such as Kernel#gets use to determine record boundaries. If set to nil, gets will read the entire file.
$-0	String	Synonym for $/.
$\	String	The string appended to the output of every call to methods such as Kernel#print and IO#write. The default value is nil.
$,	String	The separator string output between the parameters to methods such as Kernel#print and Array#join. Defaults to nil, which adds no text.
$.	Fixnum	The number of the last line read from the current input file.

$;	String	The default separator pattern used by String#split. May be set from the command line using the -F flag.
$<	Object	An object that provides access to the concatenation of the contents of all the files given as command-line arguments or $stdin (in the case where there are no arguments). $< supports methods similar to a File object: binmode, close, closed?, each, each_byte, each_line, eof, eof?, file, filename, fileno, getc, gets, lineno, lineno=, path, pos, pos=, read, readchar, readline, readlines, rewind, seek, skip, tell, to_a, to_i, to_io, to_s, along with the methods in Enumerable. The method file returns a File object for the file currently being read. This may change as $< reads through the files on the command line. [r/o]
$>	IO	The destination of output for Kernel#print and Kernel#printf. The default value is $stdout.
$_	String	The last line read by Kernel#gets or Kernel#readline. Many string-related functions in the Kernel module operate on $_ by default. The variable is local to the current scope. [thread]
$-F	String	Synonym for $;.
$stderr	IO	The current standard error output.
$stdin	IO	The current standard input.
$stdout	IO	The current standard output. Assignment to $stdout is not permitted: use $stdout.reopen instead.

1.9 The variables $defout and $deferr have been removed from Ruby 1.9.

Execution Environment Variables

$0	String	The name of the top-level Ruby program being executed. Typically this will be the program's filename. On some operating systems, assigning to this variable will change the name of the process reported (for example) by the ps(1) command.
$*	Array	An array of strings containing the command-line options from the invocation of the program. Options used by the Ruby interpreter will have been removed. [r/o]
$"	Array	An array containing the filenames of modules loaded by require. [r/o]
$$	Fixnum	The process number of the program being executed. [r/o]
$?	Process::Status	The exit status of the last child process to terminate. [r/o, thread]
$:	Array	An array of strings, where each string specifies a directory to be searched for Ruby scripts and binary extensions used by the load and require methods. The initial value is the value of the arguments passed via the -I command-line option, followed by an installation-defined standard library location, followed by the current directory (“.”). This variable may be set from within a program to alter the default search path; typically, programs use $: << dir to append dir to the path. [r/o]
$-a	Object	True if the -a option is specified on the command line. [r/o]
1.9 __callee__	Symbol	The name of the lexically enclosing method.
$-d	Object	Synonym for $DEBUG.
$DEBUG	Object	Set to true if the -d command-line option is specified.

__ENCODING__ (1.9)	String	The encoding of the current source file. [r/o]
__FILE__	String	The name of the current source file. [r/o]
$F	Array	The array that receives the split input line if the -a command-line option is used.
$FILENAME	String	The name of the current input file. Equivalent to $<.filename. [r/o]
$-i	String	If in-place edit mode is enabled (perhaps using the -i command-line option), $-i holds the extension used when creating the backup file. If you set a value into $-i, enables in-place edit mode. See page 235.
$-I	Array	Synonym for $:. [r/o]
$-l	Object	Set to true if the -l option (which enables line-end processing) is present on the command line. See page 235. [r/o]
__LINE__	String	The current line number in the source file. [r/o]
$LOAD_PATH	Array	A synonym for $:. [r/o]
$LOADED_FEATURES	Array	Synonym for $". [r/o]
__method__ (1.9)	Symbol	The name of the lexically enclosing method.
$PROGRAM_NAME	String	Alias for $0.
$-p	Object	Set to true if the -p option (which puts an implicit while gets ... end loop around your program) is present on the command line. See page 235. [r/o]
$SAFE	Fixnum	The current safe level (see page 437). This variable's value may never be reduced by assignment. [thread]
$VERBOSE	Object	Set to true if the -v, --version, -W, or -w option is specified on the command line. Set to false if no option, or -W1 is given. Set to nil if -W0 was specified. Setting this option to true causes the interpreter and some library routines to report additional information. Setting to nil suppresses all warnings (including the output of Kernel.warn).
$-v	Object	Synonym for $VERBOSE.
$-w	Object	Synonym for $VERBOSE.
$-W	Object	Return the value set by the -W command-line option.

Standard Objects

ARGF	Object	A synonym for $<.
ARGV	Array	A synonym for $*.
ENV	Object	A hash-like object containing the program's environment variables. An instance of class Object, ENV implements the full set of Hash methods. Used to query and set the value of an environment variable, as in ENV["PATH"] and ENV["term"]="ansi".
false	FalseClass	Singleton instance of class FalseClass. [r/o]

nil	NilClass	The singleton instance of class NilClass. The value of uninitialized instance and global variables. [r/o]
self	Object	The receiver (object) of the current method. [r/o]
true	TrueClass	Singleton instance of class TrueClass. [r/o]

Global Constants

The following constants are defined by the Ruby interpreter.

	DATA	IO	If the main program file contains the directive __END__, then the constant DATA will be initialized so that reading from it will return lines following __END__ from the source file.
	FALSE	FalseClass	Constant containing reference to false.
	NIL	NilClass	Constant containing reference to nil.
	RUBY_COPYRIGHT	String	The interpreter copyright.
1.9	RUBY_DESCRIPTION	String	Version number and architecture of the interpreter.
1.9	RUBY_ENGINE	String	The name of the Ruby interpreter. Returns ruby for Matz's version. Other interpreters include macruby, ironruby, jruby, and rubinius.
1.9	RUBY_PATCHLEVEL	String	The patch level of the interpreter.
1.9	RUBY_PLATFORM	String	The identifier of the platform running this program. This string is in the same form as the platform identifier used by the GNU configure utility (which is not a coincidence).
	RUBY_RELEASE_DATE	String	The date of this release.
	RUBY_REVISION	String	The revision of the interpreter.
	RUBY_VERSION	String	The version number of the interpreter.
	STDERR	IO	The actual standard error stream for the program. The initial value of $stderr.
	STDIN	IO	The actual standard input stream for the program. The initial value of $stdin.
	STDOUT	IO	The actual standard output stream for the program. The initial value of $stdout.
	SCRIPT_LINES__	Hash	If a constant SCRIPT_LINES__ is defined and references a Hash, Ruby will store an entry containing the contents of each file it parses, with the file's name as the key and an array of strings as the value. See Kernel.require on page 576 for an example.
	TOPLEVEL_BINDING	Binding	A Binding object representing the binding at Ruby's top level—the level where programs are initially executed.
	TRUE	TrueClass	A reference to the object true.

The constant __FILE__ and the variable $0 are often used together to run code only if it appears in the file run directly by the user. For example, library writers often use this to

include tests in their libraries that will be run if the library source is run directly, but not if the source is required into another program.

```
# library code
# ...
if __FILE__ == $0
  # tests...
end
```

Expressions

Single terms in an expression may be any of the following.

- **Literal**. Ruby literals are numbers, strings, arrays, hashes, ranges, symbols, and regular expressions. These are described starting on page 327.
- **Shell command**. A shell command is a string enclosed in backquotes or in a general delimited string (page 326) starting with %x. The value of the string is the standard output of running the command represented by the string under the host operating system's standard shell. The execution also sets the $? variable with the command's exit status.

  ```
  filter = "*.c"
  files = `ls #{filter}`
  files = %x{ls #{filter}}
  ```

- **Variable reference** or **constant reference**. A variable is referenced by citing its name. Depending on scope (see page 336), a constant is referenced either by citing its name or by qualifying the name, using the name of the class or module containing the constant and the scope operator (::).

  ```
  barney    # variable reference
  APP_NAMR  # constant reference
  Math::PI  # qualified constant reference
  ```

- **Method invocation**. The various ways of invoking a method are described starting on page 355.

Operator Expressions

Expressions may be combined using operators. Table 22.4 on the following page lists the Ruby operators in precedence order. The operators with a ✓ in the Method column are implemented as methods and may be overridden.

More on Assignment

The assignment operator assigns one or more *rvalues* (the *r* stands for "right," because rvalues tend to appear on the right side of assignments) to one or more *lvalues* ("left" values). What is meant by assignment depends on each individual lvalue.

Table 22.4. Ruby Operators (High to Low Precedence)

Method	Operator	Description
✓	[ ] [ ]=	Element reference, element set
✓	**	Exponentiation
✓	! ~ + −	Not, complement, unary plus and minus (method names for the last two are +@ and -@)
✓	* / %	Multiply, divide, and modulo
✓	+ −	Plus and minus
✓	>> <<	Right and left shift (<< is also used as the append operator)
✓	&	"And" (bitwise for integers)
✓	^ |	Exclusive "or" and regular "or" (bitwise for integers)
✓	<= < > >=	Comparison operators
✓	<=> == === != =~ !~	Equality and pattern match operators
	&&	Logical "and"
	||	Logical "or"
	.. ...	Range (inclusive and exclusive)
	? :	Ternary if-then-else
	= %= /= −= += |= &= >>= <<= *= &&= ||= **=	Assignment
	not	Logical negation
	or and	Logical composition
	if unless while until	Expression modifiers
	begin/end	Block expression

If an lvalue is a variable or constant name, that variable or constant receives a reference to the corresponding rvalue:

```
a = /regexp/
b, c, d = 1, "cat", [ 3, 4, 5 ]
```

If the lvalue is an object attribute, the corresponding attribute setting method will be called in the receiver, passing as a parameter the rvalue:

```
obj = A.new
obj.value = "hello"   # equivalent to obj.value=("hello")
```

If the lvalue is an array element reference, Ruby calls the element assignment operator ([]=) in the receiver, passing as parameters any indices that appear between the brackets followed by the rvalue. This is illustrated in the following table.

Element Reference	Actual Method Call
var[ ] = "one"	var.[ ]=("one")
var[1] = "two"	var.[ ]=(1, "two")
var["a", /^cat/] = "three"	var.[ ]=("a", /^cat/, "three")

If you are writing an [ ]= method that accepts a variable number of indices, it might be convenient to define it using this:

```
def []=(*indices, value)
  # ...
end
```

The value of an assignment expression is its rvalue. This is true even if the assignment is to an attribute method that returns something different.

Parallel Assignment

An assignment expression may have one or more lvalues and one or more rvalues. This section explains how Ruby handles assignment with different combinations of arguments:

1. If any rvalue is prefixed with an asterisk and implements to_a, the rvalue is replaced with the elements returned by to_a, with each element forming its own rvalue.
2. If the assignment contains one lvalue and multiple rvalues, the rvalues are converted to an array and assigned to that lvalue.
3. If the assignment contains multiple lvalues and one rvalue, the rvalue is expanded if possible into a set of rvalues as described in (1).
4. Successive rvalues are assigned to the lvalues. This assignment effectively happens in parallel, so that (for example) a,b=b,a swaps the values in a and b.
5. If there are more lvalues than rvalues, the excess will have nil assigned to them.
6. If there are more rvalues than lvalues, the excess will be ignored.
7. At most one lvalue can be prefixed by an asterisk. This lvalue will end up being an array and will contain as many rvalues as possible. If there are lvalues to the right of the starred lvalue, these will be assigned from the trailing rvalues, and whatever rvalues are left will be assigned to the splat lvalue.
8. If an lvalue contains a parenthesized list, the list is treated as a nested assignment statement, and then it is assigned from the corresponding rvalue as described by these rules.

The tutorial has examples starting on page 151. The value of a parallel assignment is its set of rvalues.

Block Expressions

```
begin
  body
end
```

Expressions may be grouped between begin and end. The value of the block expression is the value of the last expression executed.

Block expressions also play a role in exception handling, which is discussed starting on page 367.

Boolean Expressions

Ruby predefines the globals false and nil. Both of these values are treated as being false in a boolean context. All other values are treated as being true. The constant true is available for when you need an explicit "true" value.

And, Or, Not

The and and && operators evaluate their first operand. If false, the expression returns the value of the first operand; otherwise, the expression returns the value of the second operand:

expr1 `and` *expr2*
expr1 `&&` *expr2*

The or and || operators evaluate their first operand. If true, the expression returns the value of their first operand; otherwise, the expression returns the value of the second operand:

expr1 `or` *expr2*
expr1 `||` *expr2*

The not and ! operators evaluate their operand. If true, the expression returns false. If false, the expression returns true. For historical reasons, a string, regexp, or range may not appear as the single argument to not or !.

The word forms of these operators (and, or, and not) have a lower precedence than the corresponding symbol forms (&&, ||, and !). See Table 22.4 on page 345 for details.

defined?

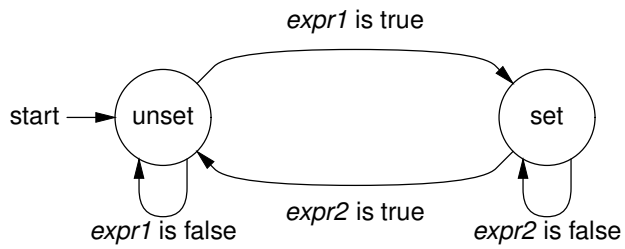
The defined? keyword returns nil if its argument, which can be an arbitrary expression, is not defined. Otherwise, it returns a description of that argument. For examples, see page 154 in the tutorial.

Comparison Operators

The Ruby syntax defines the comparison operators ==, ===, <=>, <, <=, >, >=, =~. All of these operators are implemented as methods. By convention, the language also uses the standard methods eql? and equal? (see Table 9.1 on page 156). Although the operators have intuitive meaning, it is up to the classes that implement them to produce meaningful comparison semantics. The library reference starting on page 442 describes the comparison semantics for the built-in classes. The module Comparable provides support for implementing the operators ==, <, <=, >, and >=, as well as the method between? in terms of <=>. The operator === is used in case expressions, described on page 349.

1.9 Both == and =~ have negated forms, != and !~. If an object defines these methods, Ruby will call them. Otherwise, a != b is mapped to !(a == b), and a !~ b is mapped to !(a =~ b).

Figure 22.1. State Transitions for Boolean Range



Ranges in Boolean Expressions

```
if  expr1 .. expr2
while  expr1 ... expr2
```

A range used in a boolean expression acts as a flip-flop. It has two states, set and unset, and is initially unset. On each call, the range executes a transition in the state machine shown in Figure 22.1. The range expression returns true if the state machine is in the set state at the end of the call, and false otherwise.

The two-dot form of a range behaves slightly differently than the three-dot form. When the two-dot form first makes the transition from unset to set, it immediately evaluates the end condition and makes the transition accordingly. This means that if *expr1* and *expr2* both evaluate to true on the same call, the two-dot form will finish the call in the unset state. However, it still returns true for this call.

The three-dot form does not evaluate the end condition immediately upon entering the set state.

The difference is illustrated by the following code:

```
a = (11..20).collect {|i| (i%4 == 0)..(i%3 == 0) ? i : nil}
a   # =>   [nil, 12, nil, nil, nil, 16, 17, 18, nil, 20]

a = (11..20).collect {|i| (i%4 == 0)...(i%3 == 0) ? i : nil}
a   # =>   [nil, 12, 13, 14, 15, 16, 17, 18, nil, 20]
```

Regular Expressions in Boolean Expressions

1.9

In versions of Ruby prior to 1.8, a single regular expression in boolean expression was matched against the current value of the variable $_. This behavior is now supported only if the condition appears in a command-line -e parameter:

```
$ ruby -ne 'print if /one/' testfile
```

In regular code, the use of implicit operands and $_ is being slowly phased out, so it is better to use an explicit match against a variable. If a match against $_ is required, use this:

```
if ~/re/ ...      or       if $_ =~ /re/ ...
```

`if` and `unless` Expressions

```
if boolean-expression [ then ]
  body
[ elsif boolean-expression [ then ]
  body , ... ]
[  else
  body  ]
end
```

```
unless boolean-expression [ then ]
  body
[ else
  body  ]
end
```

The then keyword separates the body from the condition.[6] It is not required if the body starts on a new line. The value of an if or unless expression is the value of the last expression evaluated in whichever body is executed.

`if` and `unless` Modifiers

```
expression if      boolean-expression
expression unless  boolean-expression
```

This evaluates *expression* only if *boolean-expression* is true (for if) or false (for unless).

Ternary Operator

```
boolean-expression ? expr1 : expr2
```

This returns *expr1* if *boolean expression* is true and *expr2* otherwise.

`case` Expressions

Ruby has two forms of case statement. The first allows a series of conditions to be evaluated, executing code corresponding to the first condition that is true:

```
case
when condition [, condition ]... [ then ]
    body
when condition [, condition ]... [ then ]
    body
  ...
[ else
    body  ]
end
```

6. Prior to Ruby 1.9, you could use a colon instead of then. This is no longer supported.

The second form of a case expression takes a target expression following the case keyword. It searches for a match by starting at the first (top left) comparison, performing *comparison* === *target*:

```
case target
when comparison [, comparison ]... [ then ]
    body
when comparison [, comparison ]... [ then ]
    body
  ...
[ else
    body  ]
end
```

A comparison can be an array reference preceded by an asterisk, in which case it is expanded into that array's elements before the tests are performed on each. When a comparison returns true, the search stops, and the body associated with the comparison is executed (no break is required). case then returns the value of the last expression executed. If no *comparison* matches, this happens: if an else clause is present, its body will be executed; otherwise, case silently returns nil.

The then keyword separates the when comparisons from the bodies and is not needed if the body starts on a new line.

1.9 As an optimization in Matz's Ruby 1.9, comparisons with literal strings and numbers do not use ===.

Loops

```
while boolean-expression [ do ]
  body
end
```

This executes *body* zero or more times as long as *boolean-expression* is true.

```
until boolean-expression [ do ]
  body
end
```

This executes *body* zero or more times as long as *boolean-expression* is false.

In both forms, the do separates *boolean-expression* from the *body* and can be omitted when the body starts on a new line:

```
for name [, name ]... in expression [ do ]
  body
end
```

The for loop is executed as if it were the following each loop, except that local variables defined in the body of the for loop will be available outside the loop, and those defined within an iterator block will not.

```
expression.each do | name [, name ]... |
  body
end
```

loop, which iterates its associated block, is not a language construct—it is a method in module Kernel. *Library*

```
loop do
  print "Input: "
  break unless line = gets
  process(line)
end
```

while and until Modifiers

expression `while` *boolean-expression*
expression `until` *boolean-expression*

If *expression* is anything other than a begin/end block, executes *expression* zero or more times while *boolean-expression* is true (for while) or false (for until).

If *expression* is a begin/end block, the block will always be executed at least one time.

break, redo, next, and retry

break, redo, next, and retry alter the normal flow through a while, until, for, or iterator controlled loop.

break terminates the immediately enclosing loop—control resumes at the statement following the block. redo repeats the loop from the start but without reevaluating the condition or fetching the next element (in an iterator). The next keyword skips to the end of the loop, effectively starting the next iteration. retry restarts the loop, reevaluating the condition.

break and next may optionally take one or more arguments. If used within a block, the given argument(s) are returned as the value of the yield. If used within a while, until, or for loop, the value given to break is returned as the value of the statement, and the value given to next is silently ignored. If break is never called or if it is called with no value, the loop returns nil.

```
match = while line = gets
          next if line =~ /^#/
          break line if line =~ /ruby/
        end

match = for line in ARGF.readlines
          next if line =~ /^#/
          break line if line =~ /ruby/
        end
```

Method Definition

```
def defname [ ( [ arg [ =val ], ... ] [ , &blockarg ] ) ]
  body
end
```

defname is both the name of the method and optionally the context in which it is valid.

defname ← *methodname*
constant.*methodname*
(*expr*).*methodname*

A *methodname* is either a redefinable operator (see Table 22.4 on page 345) or a name. If *methodname* is a name, it should start with a lowercase letter (or underscore) optionally followed by uppercase and lowercase letters, underscores, and digits. A *methodname* may optionally end with a question mark (?), exclamation point (!), or equals sign (=). The question mark and exclamation point are simply part of the name. The equals sign is also part of the name but additionally signals that this method may be used as an lvalue (described on page 55).

A method definition using an unadorned method name within a class or module definition creates an instance method. An instance method may be invoked only by sending its name to a receiver that is an instance of the class that defined it (or one of that class's subclasses).

Outside a class or module definition, a definition with an unadorned method name is added as a private method to class Object and hence may be called in any context without an explicit receiver.

A definition using a method name of the form *constant.methodname* or the more general (*expr*).*methodname* creates a method associated with the object that is the value of the constant or expression; the method will be callable only by supplying the object referenced by the expression as a receiver. This style of definition creates per object or *singleton methods*.

```
class MyClass
  def MyClass.method      # definition
  end
end
MyClass.method            # call
obj = Object.new
def obj.method            # definition
end
obj.method                # call
def (1.class).fred        # receiver may be an expression
end
Fixnum.fred               # call
```

Method definitions may not contain class or module definitions. They may contain nested instance or singleton method definitions. The internal method is defined when the enclosing method is executed. The internal method does *not* act as a closure in the context of the nested method—it is self-contained.

```
def toggle
  def toggle
    "subsequent times"
  end
  "first time"
end

toggle   # =>   "first time"
toggle   # =>   "subsequent times"
toggle   # =>   "subsequent times"
```

The body of a method acts as if it were a begin/end block, in that it may contain exception handling statements (rescue, else, and ensure).

Method Arguments

A method definition may have zero or more regular arguments and an optional block argument. Arguments are separated by commas, and the argument list may be enclosed in parentheses.

A regular argument is a local variable name, optionally followed by an equals sign and an expression giving a default value. The expression is evaluated at the time the method is called. The expressions are evaluated from left to right. An expression may reference a parameter that precedes it in the argument list.

```
def options(a=99, b=a+1)
  [ a, b ]
end
options        # =>   [99, 100]
options 1      # =>   [1, 2]
options 2, 4   # =>   [2, 4]
```

1.9 In Ruby 1.9, arguments without default values may appear after arguments with defaults. When such a method is called, Ruby will use the default values only if fewer parameters are passed to the method call than the total number of arguments.

```
def mixed(a, b=50, c=b+10, d)
  [ a, b, c, d ]
end
mixed 1, 2         # =>   [1, 50, 60, 2]
mixed 1, 2, 3      # =>   [1, 2, 12, 3]
mixed 1, 2, 3, 4   # =>   [1, 2, 3, 4]
```

As with parallel assignment, one of the arguments may start with an asterisk. If the method call specifies any parameters in excess of the regular argument count, all these extra parameters will be collected into this newly created array.

```
def varargs(a, *b)
  [ a, b ]
end
varargs 1         # =>   [1, []]
varargs 1, 2      # =>   [1, [2]]
varargs 1, 2, 3   # =>   [1, [2, 3]]
```

1.9 In Ruby 1.9, this argument need not be the last in the argument list. See the description of parallel assignment to see how values are assigned to this parameter.

```
def splat(a, *b, c)
  [ a, b, c ]
end
splat 1, 2         # =>   [1, [], 2]
splat 1, 2, 3      # =>   [1, [2], 3]
splat 1, 2, 3, 4   # =>   [1, [2, 3], 4]
```

If an array argument follows arguments with default values, parameters will first be used to override the defaults. The remainder will then be used to populate the array.

```
def mixed(a, b=99, *c)
  [ a, b, c]
end
mixed 1            # =>   [1, 99, []]
mixed 1, 2         # =>   [1, 2, []]
mixed 1, 2, 3      # =>   [1, 2, [3]]
mixed 1, 2, 3, 4   # =>   [1, 2, [3, 4]]
```

The optional block argument must be the last in the list. Whenever the method is called, Ruby checks for an associated block. If a block is present, it is converted to an object of class Proc and assigned to the block argument. If no block is present, the argument is set to nil.

```
def example(&block)
  puts block.inspect
end
example
example { "a block" }
```

produces:

```
nil
#<Proc:0x0a5064@/tmp/prog.rb:6>
```

Undefining a Method

The keyword undef allows you to undefine a method.

```
undef name | symbol  [ , ... ]
```

An undefined method still exists—it is simply marked as being undefined. If you undefine a method in a child class and then call that method on an instance of that child class, Ruby will immediately raise a NoMethodError—it will not look for the method in the child's parents.

Invoking a Method

[receiver.] name [parameters] [block]
[receiver::] name [parameters] [block]

parameters ← (*[param, ...] [, hashlist] [*array] [&a_proc]*)

block ← { *blockbody* }
do *blockbody* end

The parentheses around the parameters may be omitted if it is otherwise unambiguous.

1.9 Initial parameters are assigned to the actual arguments of the method. Following these parameters may be a list of *key* => *value* or *key: value* pairs. These pairs are collected into a single new Hash object and passed as a single parameter.

1.9 Any parameter may be a single parameter prefixed with an asterisk. If a starred parameter supports the to_a method, that method is called, and the resulting array is expanded inline to provide parameters to the method call. If a starred argument does not support to_a, it is simply passed through unaltered.

```
def regular(a, b, *c)
  "a=#{a}, b=#{b}, c=#{c}"
end
regular 1, 2, 3, 4                      # =>   a=1, b=2, c=[3, 4]
regular(1, 2, 3, 4)                     # =>   a=1, b=2, c=[3, 4]
regular(1, *[2, 3, 4])                  # =>   a=1, b=2, c=[3, 4]
regular(1, *[2, 3], 4)                  # =>   a=1, b=2, c=[3, 4]
regular(1, *[2, 3], *4)                 # =>   a=1, b=2, c=[3, 4]
regular(*[], 1, *[], *[2, 3], *[], 4)   # =>   a=1, b=2, c=[3, 4]
```

A block may be associated with a method call using either a literal block (which must start on the same source line as the last line of the method call) or a parameter containing a reference to a Proc or Method object prefixed with an ampersand character.

```
def some_method
  yield
end
some_method { }
some_method do
end
a_proc = lambda { 99 }
some_method(&a_proc)
```

Ruby arranges for the value of Kernel.block_given? to reflect the availability of a block associated with the call, regardless of the presence of a block argument. A block argument will be set to nil if no block is specified on the call to a method.

```
def other_method(&block)
  puts "block_given = #{block_given?}, block = #{block.inspect}"
end
other_method { }
other_method
```

produces:

```
block_given = true, block = #<Proc:0x0a4f88@/tmp/prog.rb:4>
block_given = false, block = nil
```

A method is called by passing its name to a receiver. If no receiver is specified, self is assumed. The receiver checks for the method definition in its own class and then sequentially in its ancestor classes. The instance methods of included modules act as if they were in anonymous superclasses of the class that includes them. If the method is not found, Ruby invokes the method method_missing in the receiver. The default behavior defined in Kernel.method_missing is to report an error and terminate the program. *Library*

When a receiver is explicitly specified in a method invocation, it may be separated from the method name using either a period (.) or two colons (::). The only difference between these two forms occurs if the method name starts with an uppercase letter. In this case, Ruby will assume that a receiver::Thing method call is actually an attempt to access a constant called Thing in the receiver *unless* the method invocation has a parameter list between parentheses. Using :: to indicate a method call is mildly deprecated.

```
Foo.Bar()         #  method call
Foo.Bar           #  method call
Foo::Bar()        #  method call
Foo::Bar          #  constant access
```

The return value of a method is the value of the last expression executed.

```
def odd_or_even(val)
  if val.odd?
    "odd"
  else
    "even"
  end
end

odd_or_even(26)   # =>   "even"
odd_or_even(27)   # =>   "odd"
```

A return expression immediately exits a method.

```
return [ expr, ... ]
```

The value of a return is nil if it is called with no parameters, the value of its parameter if it is called with one parameter, or an array containing all of its parameters if it is called with more than one parameter.

super

```
super  [  ( [ param, ... ] [ *array ] )  ]  [ block ]
```

Within the body of a method, a call to super acts just like a call to that original method, except that the search for a method body starts in the superclass of the object that was found to contain the original method. If no parameters (and no parentheses) are passed to super,

the original method's parameters will be passed; otherwise, the parameters to super will be passed.

Operator Methods

expr1 operator
operator expr1
expr1 operator expr2

If the operator in an operator expression corresponds to a redefinable method (see Table 22.4 on page 345), Ruby will execute the operator expression as if it had been written like this:

`(`*expr1*`).`*operator*`()` *or*
`(`*expr1*`).`*operator*`(`*expr2*`)`

Attribute Assignment

receiver`.`*attrname* `=` *rvalue*

When the form *receiver.attrname* appears as an lvalue, Ruby invokes a method named *attrname=* in the receiver, passing *rvalue* as a single parameter. The value returned by this assignment is always *rvalue*—the return value of the method *attrname=* is discarded. If you want to access the return value (in the unlikely event that it isn't the *rvalue* anyway), send an explicit message to the method.

```
class Demo
  attr_reader :attr
  def attr=(val)
    @attr = val
    "return value"
  end
end

d = Demo.new

# In all these cases, @attr is set to 99
d.attr = 99          # =>   99
d.attr=(99)          # =>   99
d.send(:attr=, 99)   # =>   "return value"
d.attr               # =>   99
```

Element Reference Operator

receiver`[` *expr [, expr]...* `]`
receiver`[` *expr [, expr]...* `] =` *rvalue*

When used as an rvalue, element reference invokes the method [] in the receiver, passing as parameters the expressions between the brackets.

When used as an lvalue, element reference invokes the method []= in the receiver, passing as parameters the expressions between the brackets, followed by the *rvalue* being assigned.

Aliasing

```
alias new_name old_name
```

This creates a new name that refers to an existing method, operator, global variable, or regular expression backreference ($&, $`, $', and $+). Local variables, instance variables, class variables, and constants may not be aliased. The parameters to alias may be names or symbols.

```
class Fixnum
  alias plus +
end
1.plus(3)         # =>   4

alias $prematch $`
"string" =~ /i/   # =>   3
$prematch         # =>   "str"

alias :cmd :`
cmd "date"        # =>   "Mon Apr 13 13:26:12 CDT 2009\n"
```

When a method is aliased, the new name refers to a copy of the original method's body. If the method is subsequently redefined, the aliased name will still invoke the original implementation.

```
def meth
  "original method"
end
alias original meth
def meth
  "new and improved"
end
meth       # =>   "new and improved"
original   # =>   "original method"
```

Class Definition

```
class [ scope:: ] classname   [ < superexpr ]
  body
end

class << obj
  body
end
```

A Ruby class definition creates or extends an object of class Class by executing the code in *body*. In the first form, a named class is created or extended. The resulting Class object is assigned to a constant named *classname* (keep reading for scoping rules). This name should start with an uppercase letter. In the second form, an anonymous (singleton) class is associated with the specific object.

If present, *superexpr* should be an expression that evaluates to a Class object that will be the superclass of the class being defined. If omitted, it defaults to class Object.

Within *body*, most Ruby expressions are executed as the definition is read. However:

- Method definitions will register the methods in a table in the class object.
- Nested class and module definitions will be stored in constants within the class, not as global constants. These nested classes and modules can be accessed from outside the defining class using :: to qualify their names.

```
module NameSpace
  class Example
    CONST = 123
  end
end
obj = NameSpace::Example.new
a = NameSpace::Example::CONST
```

- The Module#include method will add the named modules as anonymous superclasses of the class being defined.

The *classname* in a class definition may be prefixed by the names of existing classes or modules using the scope operator (::). This syntax inserts the new definition into the namespace of the prefixing module(s) and/or class(es) but does not interpret the definition in the scope of these outer classes. A *classname* with a leading scope operator places that class or module in the top-level scope.

In the following example, class C is inserted into module A's namespace but is not interpreted in the context of A. As a result, the reference to CONST resolves to the top-level constant of that name, not A's version. We also have to fully qualify the singleton method name, because C on its own is not a known constant in the context of A::C.

```
CONST = "outer"

module A
  CONST = "inner"   # This is A::CONST
end

module A
  class B
    def B.get_const
      CONST
    end
  end
end

A::B.get_const   # =>   "inner"
```

```
class A::C
  def (A::C).get_const
    CONST
  end
end

A::C.get_const   # =>   "outer"
```

It is worth emphasizing that a class definition is executable code. Many of the directives used in class definitions (such as attr and include) are actually simply private instance methods of class Module (documented starting on page 605). The value of a class definition is the value of the last executed statement.

Chapter 24, which begins on page 384, describes in more detail how Class objects interact with the rest of the environment.

Creating Objects from Classes

obj = *classexpr*.new *[([args, ...])]*

Class Class defines the instance method Class#new, which creates an object of the class of the receiver (*classexpr* in the syntax example). This is done by calling the method *classexpr*.allocate. You can override this method, but your implementation must return an object of the correct class. It then invokes initialize in the newly created object and passes it any arguments originally passed to new.

If a class definition overrides the class method new without calling super, no objects of that class can be created, and calls to new will silently return nil.

Like any other method, initialize should call super if it wants to ensure that parent classes have been properly initialized. This is not necessary when the parent is Object, because class Object does no instance-specific initialization.

Class Attribute Declarations

Class attribute declarations are not part of the Ruby syntax; they are simply methods defined in class Module that create accessor methods automatically. *Library*

```
class name
  attr attribute  [ , writable ]
  attr_reader      attribute [, attribute ]...
  attr_writer      attribute [, attribute ]...
  attr_accessor    attribute [, attribute ]...
end
```

Module Definitions

```
module name
  body
end
```

A module is basically a class that cannot be instantiated. Like a class, its body is executed during definition, and the resulting Module object is stored in a constant. A module may contain class and instance methods and may define constants and class variables. As with classes, module methods are invoked using the Module object as a receiver, and constants are accessed using the :: scope resolution operator. The name in a module definition may optionally be preceded by the names of enclosing class(es) and/or module(s).

```
CONST = "outer"
module Mod
  CONST = 1
  def Mod.method1    # module method
    CONST + 1
  end
end
module Mod::Inner
  def (Mod::Inner).method2
    CONST + " scope"
  end
end

Mod::CONST            # =>   1
Mod.method1           # =>   2
Mod::Inner::method2   # =>   "outer scope"
```

Mixins: Including Modules

```
class|module name
  include expr
end
```

A module may be included within the definition of another module or class using the include method. The module or class definition containing the include gains access to the constants, class variables, and instance methods of the module it includes. *Library*

If a module is included within a class definition, the module's constants, class variables, and instance methods made available via an anonymous (and inaccessible) superclass for that class. Objects of the class will respond to messages sent to the module's instance methods. Calls to methods not defined in the class will be passed to the module(s) mixed into the class before being passed to any parent class. A module may choose to define an initialize method, which will be called upon the creation of an object of a class that mixes in the module if either (a) the class does not define its own initialize method or (b) the class's initialize method invokes super.

A module may also be included at the top level, in which case the module's constants, class variables, and instance methods become available at the top level.

Module Functions

Although include is useful for providing mixin functionality, it is also a way of bringing the constants, class variables, and instance methods of a module into another namespace. However, functionality defined in an instance method will not be available as a module method.

```
module Math
  def sin(x)
    #
  end
end
# Only way to access Math.sin is...
include Math
sin(1)
```

The method Module#module_function solves this problem by taking one or more module instance methods and copying their definitions into corresponding module methods. *Library*

```
module Math
  def sin(x)
    #
  end
  module_function :sin
end
Math.sin(1)
include Math
sin(1)
```

The instance method and module method are two different methods: the method definition is copied by module_function, not aliased.

You can also use module_function with no parameters, in which case all subsequent methods will be module methods.

Access Control

Ruby defines three levels of protection for module and class constants and methods:

- **Public**. Accessible to anyone.
- **Protected**. Can be invoked only by objects of the defining class and its subclasses.
- **Private**. Can be called only in functional form (that is, with an implicit self as the receiver). Private methods therefore can be called in the defining class and by that class's descendents and ancestors, but only within the same object. See the discussion starting on page 61 for examples.

```
private    [ symbol, ... ]
protected  [ symbol, ... ]
public     [ symbol, ... ]
```

Each function can be used in two different ways: *Library*

- If used with no arguments, the three functions set the default access control of subsequently defined methods.
- With arguments, the functions set the access control of the named methods and constants.

Access control is enforced when a method is invoked.

Blocks, Closures, and Proc Objects

A code block is a set of Ruby statements and expressions between braces or a do/end pair. The block may start with an argument list between vertical bars. A code block may appear only immediately after a method invocation. The start of the block (the brace or the do) must be on the same logical line as the end of the invocation.

```
invocation  do  | a1, a2, ... |
end

invocation  {   | a1, a2, ... |
}
```

Braces have a high precedence; do has a low precedence. If the method invocation has parameters that are not enclosed in parentheses, the brace form of a block will bind to the last parameter, not to the overall invocation. The do form will bind to the invocation.

Within the body of the invoked method, the code block may be called using the yield keyword. Parameters passed to the yield will be assigned to arguments in the block. A warning will be generated if yield passes multiple parameters to a block that takes just one. The return value of the yield is the value of the last expression evaluated in the block or the value passed to a next statement executed in the block.

A block is a *closure*; it remembers the context in which it was defined, and it uses that context whenever it is called. The context includes the value of *self*, the constants, class variables, local variables, and any captured block.

```
class BlockExample
  CONST = 0
  @@a = 3
  def return_closure
    a = 1
    @a = 2
    lambda { [ CONST, a, @a, @@a, yield ] }
  end
  def change_values
    @a += 1
    @@a += 1
  end
end

eg = BlockExample.new
block = eg.return_closure { "original" }

block.call   # =>  [0, 1, 2, 3, "original"]
eg.change_values
block.call   # =>  [0, 1, 3, 4, "original"]
```

Here, the return_closure method returns a lambda that encapsulates access to the local variable a, instance variable @a, class variable @@a, and constant CONST. We call the block outside the scope of the object that contains these values, and they are still available via the

closure. If we then call the object to change some of the values, the values accessed via the closure also change.

Block Arguments

1.9 As of Ruby 1.9, block argument lists are more like method argument lists:

- You can specify default values.
- You can specify splat (starred) arguments.
- The last argument can be prefixed with an ampersand, in which case it will collect any block passed when the original block is called.

These changes make it possible to use Module#define_method to create methods based on blocks that have similar capabilities to methods created using def.

Proc Objects

Ruby's blocks are chunks of code attached to a method. They operate in the context in which they were defined. Blocks are not objects, but they can be converted into objects of class Proc. There are four ways of converting a block into a Proc object.

- By passing a block to a method whose last parameter is prefixed with an ampersand. That parameter will receive the block as a Proc object.

```
def meth1(p1, p2, &block)
  puts block.inspect
end
meth1(1,2) { "a block" }
meth1(3,4)
```

produces:

```
#<Proc:0x0a4f4c@/tmp/prog.rb:4>
nil
```

- By calling Proc.new, again associating it with a block.[7] *Library*

```
block = Proc.new { "a block" }
block   # =>   #<Proc:0x0a53c0@/tmp/prog.rb:1>
```

- By calling the method Kernel.lambda, associating a block with the call. *Library*

```
block = lambda { "a block" }
block   # =>   #<Proc:0x0a53e8@/tmp/prog.rb:1 (lambda)>
```

1.9 - As of Ruby 1.9, using the -> syntax.

```
lam = ->(p1, p2) { p1 + p2 }
lam.call(4, 3)   # =>   7
```

Note that there cannot be a space between > and the opening parenthesis.

1.9 7. There's also a built-in Kernel.proc method. In Ruby 1.8, this was equivalent to lambda. In Ruby 1.9, it is the same as Proc.new. Don't use proc in new code.

The first two styles of Proc object are identical in use. We'll call these objects *raw procs*. The third and fourth styles, generated by lambda and ->, add some functionality to the Proc object, as we'll see in a minute. We'll call these objects *lambdas*.

Calling a Proc

You can call a proc by invoking its methods call, yield, or []. The three forms are identical. Each takes arguments that can be passed to the proc, just as if it were a regular method call. If the proc you're invoking is a lambda, Ruby will check that the supplied arguments match the expected parameters.

You can also invoke a proc using the syntax name.(*args*...). This is mapped internally into a.call(...).

Procs, break, and next

Within both raw procs and lambdas, executing next causes the block to exit. The value of the block is the value (or values) passed to next, or nil if no values are passed.

```
def meth
  res = yield
  "The block returns #{res}"
end

meth { next 99 }   # =>   "The block returns 99"

pr = Proc.new { next 99 }
pr.call            # =>   99

pr = lambda { next 99 }
pr.call            # =>   99

pr = ->() { next 99 }
pr.call            # =>   99
```

Within a raw proc, a break terminates the method that invoked the block. The return value of the method is any parameters passed to the break.

Return and Blocks

A return from inside a *block* that's still in scope acts as a return from that scope. A return from a block whose original context is not longer valid raises an exception (LocalJumpError or ThreadError depending on the context). The following example illustrates the first case:

```
def meth1
  (1..10).each do |val|
    return val           # returns from meth1
  end
end
meth1   # =>   1
```

This example shows a return failing because the context of its block no longer exists:

```
def meth2(&b)
  b
end
res = meth2 { return }
res.call
```

produces:

```
prog.rb:5:in `block in <main>': unexpected return (LocalJumpError)
from /tmp/prog.rb:6:in `call'
from /tmp/prog.rb:6:in `<main>'
```

And here's a return failing because the block is created in one thread and called in another:

```
def meth3
  yield
end
t = Thread.new do
  meth3 { return }
end
t.join
```

produces:

```
prog.rb:6:in `block (2 levels) in <main>': unexpected return (LocalJumpError)
from /tmp/prog.rb:2:in `meth3'
from /tmp/prog.rb:6:in `block in <main>'
```

The situation with Proc objects is slightly more complicated. If you use Proc.new to create a proc from a block, that proc acts like a block, and the previous rules apply:

```
def meth4
  p = Proc.new { return 99 }
  p.call
  puts "Never get here"
end

meth4   # =>   99
```

If the Proc object is created using Kernel.lambda, it behaves more like a free-standing method body: a return simply returns from the block to the caller of the block:

```
def meth5
  p = lambda { return 99 }
  res = p.call
  "The block returned #{res}"
end

meth5   # =>   "The block returned 99"
```

Because of this, if you use Module#define_method, you'll probably want to pass it a proc created using lambda, not Proc.new, because return will work as expected in the former and will generate a LocalJumpError in the latter.

Exceptions

Ruby exceptions are objects of class Exception and its descendents (a full list of the built-in exceptioons is given in Figure 27.1 on page 502).

Raising Exceptions

The Kernel.raise method raises an exception: *Library*

```
raise
raise string
raise thing [ , string [ stack trace ] ]
```

The first form reraises the exception in $! or a new RuntimeError if $! is nil.

The second form creates a new RuntimeError exception, setting its message to the given string.

The third form creates an exception object by invoking the method exception on its first argument. It then sets this exception's message and backtrace to its second and third arguments.

Class Exception and objects of class Exception contain a factory method called exception, so an exception class name or instance can be used as the first parameter to raise.

When an exception is raised, Ruby places a reference to the Exception object in the global variable $!.

Handling Exceptions

Exceptions may be handled in the following ways:

- Within the scope of a begin/end block:

```
begin
  code...
  code...
[ rescue  [ parm, ... ] [ => var ] [ then ]
    error handling code... , ... ]
[ else
    no exception code... ]
[ ensure
    always executed code... ]
end
```

- Within the body of a method:

```
def method and args
  code...
  code...
[ rescue  [ parm, ... ] [ => var ] [ then ]
    error handling code... , ... ]
[ else
    no exception code... ]
```

```
[ ensure
    always executed code... ]
  end
```

- After the execution of a single statement:

```
statement [ rescue statement, ... ]
```

A block or method may have multiple rescue clauses, and each rescue clause may specify zero or more exception parameters. A rescue clause with no parameter is treated as if it had a parameter of StandardError. This means that some lower-level exceptions will not be caught by a parameterless rescue class. If you want to rescue every exception, use this:

```
rescue Exception => e
```

When an exception is raised, Ruby scans the call stack until it finds an enclosing begin/end block, method body, or statement with a rescue modifier. For each rescue clause in that block, Ruby compares the raised exception against each of the rescue clause's parameters in turn; each parameter is tested using *parameter*===$!. If the raised exception matches a rescue parameter, Ruby executes the body of the rescue and stops looking. If a matching rescue clause ends with => and a variable name, the variable is set to $!.

Although the parameters to the rescue clause are typically the names of Exception classes, they can actually be arbitrary expressions (including method calls) that return an appropriate class.

If no rescue clause matches the raised exception, Ruby moves up the stack looking for a higher-level begin/end block that matches. If an exception propagates to the top level of the main thread without being rescued, the program terminates with a message.

If an else clause is present, its body is executed if no exceptions were raised in *code*. Exceptions raised during the execution of the else clause are not captured by rescue clauses in the same block as the else.

If an ensure clause is present, its body is always executed as the block is exited (even if an uncaught exception is in the process of being propagated).

Within a rescue clause, raise with no parameters will reraise the exception in $!.

Rescue Statement Modifier

A statement may have an optional rescue modifier followed by another statement (and by extension another rescue modifier, and so on). The rescue modifier takes no exception parameter and rescues StandardError and its children.

If an exception is raised to the left of a rescue modifier, the statement on the left is abandoned, and the value of the overall line is the value of the statement on the right:

```
values = [ "1", "2.3", /pattern/ ]

result = values.map {|v| Integer(v) rescue Float(v) rescue String(v) }

result   # =>   [1, 2.3, "(?-mix:pattern)"]
```

Retrying a Block

The retry statement can be used within a rescue clause to restart the enclosing begin/end block from the beginning.

Catch and Throw

The method Kernel.catch executes its associated block: *Library*

```
catch ( symbol | string )  do
  block...
end
```

The method Kernel.throw interrupts the normal processing of statements: *Library*

```
throw( symbol | string [ , obj ] )
```

When a throw is executed, Ruby searches up the call stack for the first catch block with a matching symbol or string. If it is found, the search stops, and execution resumes past the end of the catch's block. If the throw was passed a second parameter, that value is returned as the value of the catch. Ruby honors the ensure clauses of any block expressions it traverses while looking for a corresponding catch.

If no catch block matches the throw, Ruby raises a NameError exception at the location of the throw.