

# keywords.rb

**Path:** keywords.rb

**Last Update:** Thu Oct 22 23:00:56 -0700 2009

RDoc-style documentation [for](#) Ruby keywords (1.9.1).

David A. Black

June 29, 2009

Yes, I KNOW that they aren't methods. I've just put them [in](#) that format to produce the familiar RDoc output. I've been focusing on the content.

If anyone has a good idea [for](#) how to package [and](#) distribute it, let me know. I haven't really thought it through.

Also, [if](#) you spot any errors [or](#) significant omissions, let me know. Keep [in](#) mind that I'm documenting the keywords themselves, [not](#) the entities they represent. Thus there is [not](#) full coverage of, say, what a [class](#) is, [or](#) how exceptions work.

Changes since first release:

- Added `__END__` (thanks Sven Fuchs)
- Added `'retry'` to `retry` example (thanks mathie)
- Corrected description of `when` `'rescue'` can be used (thanks Matt Neuburg)
- Added `else in rescue` context (thanks Rob Biedenharn)

## Methods

[BEGIN](#) [END](#) [\\_\\_ENCODING\\_\\_](#) [\\_\\_END\\_\\_](#) [\\_\\_FILE\\_\\_](#) [\\_\\_LINE\\_\\_](#) [alias](#) [and](#) [begin](#) [break](#) [case](#) [class](#)  
[def](#) [defined?](#) [do](#) [else](#) [elsif](#) [end](#) [ensure](#) [false](#) [for](#) [if](#) [in](#) [module](#) [next](#) [nil](#) [not](#) [or](#) [redo](#) [rescue](#)  
[retry](#) [return](#) [self](#) [super](#) [then](#) [true](#) [undef](#) [unless](#) [until](#) [when](#) [while](#) [yield](#)

## Public Instance methods

### [BEGIN](#)

Designates, via code block, code to be executed unconditionally before sequential execution of the program begins. Sometimes used to simulate forward references to methods.

```
puts times_3(gets.to_i)

BEGIN {
  def times_3(n)
    n * 3
  end
}
```

### [END](#)

Designates, via code block, code to be executed just prior to program termination.

```
END { puts "Bye!" }
```

## ENCODING

The current default encoding, as an Encoding instance.

## \_\_END\_\_

Denotes the [end](#) of the regular source code section of a program file. Lines below `__END__` will [not](#) be executed. Those lines will be available via the special filehandle DATA. The following code will print out two stanzas of personal information. Note that `__END__` has to be flush left, [and](#) has to be the only thing on its line.

```
DATA.each do |line|
  first, last, phone, email = line.split('|')
  puts <<-EOM
  First name: #{first}
  Last name:  #{last}
  Phone:      #{phone}
  Email:      #{email}
  EOM
end
__END__
David|Black|123-456-7890|dblack@...
Someone|Else|321-888-8888|someone@else
```

## FILE

The name of the file currently being executed, including path relative to the directory where the application was started up ([or](#) the current directory, [if](#) it has been changed). The current file is, [in](#) some cases, different from the startup file [for](#) the running application, which is available [in](#) the global variable `$0`.

## LINE

The line number, [in](#) the current source file, of the current line.

## [alias](#)

Creates an [alias or](#) duplicate method name [for](#) a given method. The original method continues to be accessible via the [alias](#), even [if](#) it is overridden. Takes two method-name arguments (which can be represented by strings [or](#) symbols but can also be the bare names themselves).

```
class Person
  def name=(name)
    puts "Naming your person #{name}!"
    @name = name
  end

  alias full_name= name=
end

p = Person.new
p.name = "David"      # Naming your person David!

class Person
  def name=(name)
    puts "Please use full_name="
  end
end

p.full_name = "David"  # Please use fullname=
```

## [and](#)

Boolean [and](#) operator. Differs from [in](#) that [and](#) has lower precedence. In this example:

```
puts "Hello" and "Goodbye"
```

the subexpression `puts "Hello"` is executed first, [and](#) returns [nil](#). The whole expression thus reduces to:

```
nil and "Goodbye"
```

which reduces to [nil](#). In this example, however:

```
puts "Hello" && "Goodbye"
```

the expression `"Hello" && "Goodbye"` is used as the argument to `puts`. This expression evaluates to `"Goodbye"`; therefore, the whole statement prints `"Goodbye"`.

## [begin](#)

Together with [end](#), delimits what is commonly called a ["begin"](#) block (to distinguish it from the Proc type of code block). A ["begin"](#) block allows the use of [while](#) [and](#) [until](#) [in](#) modifier position with multi-line statements:

```
begin
  i += 1
  puts i
end until i == 10
```

["Begin"](#) blocks also serve to scope exception raising [and](#) [rescue](#) operations. See [rescue](#) [for](#) examples. A ["begin"](#) block can have an [else](#) clause, which serves no purpose ([and](#) generates a warning) [unless](#) there's also a [rescue](#) clause, [in](#) which [case](#) the [else](#) clause is executed [when](#) no exception is raised.

## [break](#)

Causes unconditional termination of a code block [or](#) [while](#) [or](#) [until](#) block, with control transferred to the line after the block. If given an argument, returns that argument as the value of the terminated block.

```
result = File.open("lines.txt") do |fh|
  fh.each do |line|
    break line if my_regex.match(line)
  end
  nil
end
```

## [case](#)

The [case](#) statement operator. Case statements consist of an optional condition, which is [in](#) the position of an argument to [case](#), [and](#) zero [or](#) more [when](#) clauses. The first [when](#) clause to match the condition ([or](#) to evaluate to Boolean truth, [if](#) the condition is null) "wins", [and](#) its code stanza is executed. The value of the [case](#) statement is the value of the successful [when](#) clause, [or](#) [nil](#) [if](#) there is no such clause.

A [case](#) statement can [end](#) with an [else](#) clause. Each [when](#) statement can have multiple candidate values, separated by commas.

```
case x
when 1,2,3
  puts "1, 2, or 3"
when 10
  puts "10"
else
  puts "Some other number"
end
```

Case equality (success by a [when](#) candidate) is determined by the [case](#)-equality [or](#) "threequal" operator, `===`. The above example is equivalent to:

```
if 1 === x or 2 === x or 3 === x
  puts "1, 2, or 3"
elsif 10 === x
  puts "10"
else
  puts "Some other number"
end
```

`===` is typically overridden by classes to reflect meaningful [case](#)-statement behavior; [for](#) example, `/abc/ === "string"` checks [for](#) a pattern match from the string.

## [class](#)

Opens a [class](#) definition block. Takes either a constant name [or](#) an expression of the form `<< object`. In the latter [case](#), opens a definition block [for](#) the singleton [class](#) of object.

Classes may be opened more than once, [and](#) methods [and](#) constants added during those subsequent openings. [class](#) blocks have their own local scope; local variables [in](#) scope already are [not](#) visible inside the block, [and](#) variables created inside the block [do not](#) survive the block.

```
class Person
  def name=(name)
    @name = name
  end
end

david = Person.new
class << david
  def name=(name)
    if name == "David"
      @name = name
    else
      puts "Please don't name me other than David!"
    end
  end
end

david.name = "Joe" # Please don't name me other than David!"
joe = Person.new
joe.name = "Joe"
```

Inside a [class](#) block, [self](#) is set to the [class](#) object whose block it is. Thus it's possible to write [class](#) methods (i.e., singleton methods on [class](#) objects) by referring to [self](#):

```
class Person
  def self.species
    "Homo sapiens"
  end
end
```

## [def](#)

Paired with a terminating [end](#), constitutes a method definition. Starts a new local scope; local variables [in](#) existence [when](#) the [def](#) block is entered are [not in](#) scope [in](#) the block, [and](#) local variables created [in](#) the block [do not](#) survive beyond the block.

[def](#) can be used either with [or](#) without a specific object:

- [def](#) method\_name
- [def](#) object.singleton\_method\_name

The parameter list comes after the method name, [and](#) can ([and](#) usually is) wrapped [in](#) parentheses.

## [defined?](#)

[defined?](#) expression tests whether [or not](#) expression refers to anything recognizable (literal object, local variable that has been initialized, method name visible from the current scope, etc.). The [return](#) value is [nil](#) if the expression cannot be resolved. Otherwise, the [return](#) value provides information about the expression.

Note that the expression is [not](#) executed.

```
p defined?(def x; end)    # "expression"
x                        # error: undefined method or variable

p defined?(@x=1)         # "assignment"
p @x                    # nil
```

Assignment to a local variable will, however, have the usually result of initializing the variable to [nil](#) by virtue of the assignment expression itself:

```
p defined?(x=1)          # assignment
p x                      # nil
```

In most cases, the argument to [defined?](#) will be a single identifier:

```
def x; end
p defined?(x)            # "method"
```

## [do](#)

Paired with [end](#), can delimit a code block:

```
array.each do |element|
  puts element * 10
end
```

In this context, [do/end](#) is equivalent to curly braces, except that curly braces have higher precedence. In this example:

```
puts [1,2,3].map { |x| x * 10 }
```

the code block binds to map; thus the output is:

```
10
20
30
```

In this version, however:

```
puts [1,2,3].map do |x| x * 10 end
```

the code is interpreted as puts([1,2,3].map) [do](#) |x| x \* 10 [end](#). Since puts doesn't take a block, the block is ignored [and](#) the statement prints the value of the blockless [1,2,3].map (which returns an Enumerator).

[do](#) can also (optionally) appear at the [end](#) of a [for/in](#) statement. (See [for](#) [for](#) an example.)

## [else](#)

The [else](#) keyword denotes a final conditional branch. It appears [in](#) connection with [if](#), [unless](#), [and case](#), and [rescue](#). (In the [case](#) of [rescue](#), the [else](#) branch is executed [if](#) no exception is raised.) The [else](#) clause is always the last branch [in](#) the entire statement, except [in](#) the [case](#) of [rescue](#) where it can be followed by an [ensure](#) clause.

## [elsif](#)

Introduces a branch [in](#) a conditional ([if or unless](#)) statement. Such a statement can contain any number of [elsif](#) branches, including zero.

See [if for](#) examples.

## [end](#)

Marks the [end](#) of a [while](#), [until](#), [begin](#), [if](#), [def](#), [class](#), or other keyword-based, block-based construct.

## [ensure](#)

Marks the final, optional clause of a [begin/end](#) block, generally [in](#) cases where the block also contains a [rescue](#) clause. The code [in](#) the [ensure](#) clause is guaranteed to be executed, whether control flows to the [rescue](#) block or [not](#).

```
begin
  1/0
rescue ZeroDivisionError
  puts "Can't do that!"
ensure
  puts "That was fun!"
end
```

Output:

```
Can't do that!
That was fun!
```

If the statement `1/0` is changed to something harmless, like `1/1`, the [rescue](#) clause will [not](#) be executed but the [ensure](#) clause still will.

## [false](#)

[false](#) denotes a special object, the sole instance of `FalseClass`. [false and nil](#) are the only objects that evaluate to Boolean falsehood [in](#) Ruby (informally, that cause an [if](#) condition to fail.)

## [for](#)

A loop constructor, used with [in](#):

```
for a in [1,2,3,4,5] do
  puts a * 10
end
```

[for](#) is generally considered less idiomatic than `each`; indeed, [for](#) calls `each`, [and](#) is thus essentially a wrapper around it.

```
obj = Object.new
def obj.each
  yield 1; yield 2
end
for a in obj
  puts a
```

```
end
```

prints:

```
1  
2
```

The [do](#) keyword may optionally appear at the [end](#) of the [for](#) expression:

```
for a in array do  
  # etc.
```

[if](#)

Ruby's basic conditional statement constructor. [if](#) evaluates its argument [and](#) branches on the result. Additional branches can be added to an [if](#) statement with [else](#) [and](#) [elsif](#).

```
if m.score > n.score  
  puts "m wins!"  
elsif n.score > m.score  
  puts "n wins!"  
else  
  puts "Tie!"  
end
```

An [if](#) statement can have more than one [elsif](#) clause ([or](#) none), but can only have one [else](#) clause ([or](#) none). The [else](#) clause must come at the [end](#) of the entire statement.

[if](#) can also be used [in](#) modifier position:

```
puts "You lose" if y.score < 10
```

[then](#) may optionally follow an [if](#) condition:

```
if y.score.nil? then  
  puts "Have you even played the game?"  
end
```

[in](#)

See [for](#).

[module](#)

Opens a [module](#) definition block. Takes a constant (the name of the [module](#)) as its argument. The definition block starts a new local scope; existing variables are [not](#) visible inside the block, [and](#) local variables created [in](#) the block [do not](#) survive the [end](#) of the block.

Inside the [module](#) definition, [self](#) is set to the [module](#) object itself.

[next](#)

Bumps an iterator, [or](#) a [while](#) [or](#) [until](#) block, to the [next](#) iteration, unconditionally [and](#) without executing whatever may remain of the block.

```
[0,1,2,3,4].each do |n|  
  next unless n > 2  
  puts "Big number: #{n}"  
end
```

Output:

```
Big number: 3
Big number: 4
```

[next](#) is typically used [in](#) cases like iterating through a list of files [and](#) taking action ([or not](#)) depending on the filename.

[next](#) can take a value, which will be the value returned [for](#) the current iteration of the block.

```
sizes = [0,1,2,3,4].map do |n|
  next("big") if n > 2
  puts "Small number detected!"
  "small"
end

p sizes
```

Output:

```
Small number detected!
Small number detected!
Small number detected!
["small", "small", "small", "big", "big"]
```

[nil](#)

A special "non-object". [nil](#) is, [in](#) fact, an object (the sole instance of NilClass), but connotes absence [and](#) indeterminacy. [nil](#) [and](#) [false](#) are the only two objects [in](#) Ruby that have Boolean falsehood (informally, that cause an [if](#) condition to fail).

[nil](#) serves as the default value [for](#) uninitialized array elements [and](#) hash values ([unless](#) the default is overridden).

[not](#)

Boolean negation.

```
not true    # false
not 10      # false
not false   # true
```

Similar [in](#) effect to the negating bang (!), but has lower precedence:

```
not 3 == 4  # true; interpreted as not (3 == 4)
!3 == 4     # false; interpreted as (!3) == 4, i.e., false == 4
```

(The unary ! also differs [in](#) that it can be overridden.)

[or](#)

Boolean [or](#). Differs from || [in](#) that [or](#) has lower precedence. This code:

```
puts "Hi" or "Bye"
```

is interpreted as (puts "Hi") [or](#) "Bye". Since puts "Hi" reduces to [nil](#), the whole expression reduces to [nil](#) [or](#) "Bye" which evaluates to "Bye". (The side-effect printing of "Hi" does take place.)

This code, however:

```
puts "Hi" || "Bye"
```



is interpreted as `puts("Hi" || "Bye")`, which reduces to `puts "Hi"` (since `"Hi" || "Bye"` evaluates to `"Hi"`).

## [redo](#)

Causes unconditional re-execution of a code block, with the same parameter bindings as the current execution.

## [rescue](#)

Designates an exception-handling clause. Can occur either inside a [begin](#)<code>/<code>[end](#) block, inside a method definition (which implies [begin](#)), [or in](#) modifier position (at the [end](#) of a statement).

By default, [rescue](#) only intercepts `StandardError` [and](#) its descendants, but you can specify which exceptions you want handled, as arguments. (This technique does [not](#) work [when rescue](#) is [in](#) statement-modifier position.) Moreover, you can have more than one [rescue](#) clause, allowing [for](#) fine-grained handling of different exceptions.

In a method (note that `raise` with no argument, [in](#) a [rescue](#) clause, re-raises the exception that's being handled):

```
def file_reverser(file)
  File.open(file) {|fh| puts fh.readlines.reverse }
rescue Errno::ENOENT
  log "Tried to open non-existent file #{file}"
  raise
end
```

In a [begin/end](#) block:

```
begin
  1/0
rescue ZeroDivisionError
  puts "No way"
end
```

In statement-modifier position:

```
while true
  1/0
end rescue nil

david = Person.find(n) rescue Person.new
```

[rescue](#) (except [in](#) statement-modifier position) also takes a special argument [in](#) the following form:

```
rescue => e
```

which will assign the given local variable to the exception object, which can [then](#) be examined inside the [rescue](#) clause.

## [retry](#)

Inside a [rescue](#) clause, [retry](#) causes Ruby to [return](#) to the top of the enclosing code (the [begin](#) keyword, [or](#) top of method [or](#) block) [and](#) try executing the code again.

```
a = 0
begin
  1/a
rescue ZeroDivisionError => e
  puts e.message
  puts "Let's try that again..."
  a = 1
  retry
```

```
end
puts "That's better!"
```

## [return](#)

Inside a method definition, executes the [ensure](#) clause, [if](#) present, [and then](#) returns control to the context of the method call. Takes an optional argument (defaulting to [nil](#)), which serves as the [return](#) value of the method. Multiple values [in](#) argument position will be returned [in](#) an array.

```
def three
  return 3
ensure
  puts "Enjoy the 3!"
end

a = three    # Enjoy the 3!
puts a      # 3
```

Inside a code block, the behavior of [return](#) depends on whether [or not](#) the block constitutes the body of a regular Proc object [or](#) a lambda-style Proc object. In the [case](#) of a lambda, [return](#) causes execution of the block to terminate. In the [case](#) of a regular Proc, [return](#) attempts to [return](#) from the enclosing method. If there is no enclosing method, it's an error.

```
ruby -e 'Proc.new {return}.call'
=> -e:1:in `block in <main>': unexpected return (LocalJumpError)

ruby19 -e 'p lambda {return 3}.call'
=> 3
```

## [self](#)

[self](#) is the "current object" [and](#) the default receiver of messages (method calls) [for](#) which no explicit receiver is specified. Which object plays the role of [self](#) depends on the context.

- In a method, the object on which the method was called is [self](#)
- In a [class or module](#) definition (but outside of any method definition contained therein), [self](#) is the [class or module](#) object being defined.
- In a code block associated with a call to `class_eval` (aka `module_eval`), [self](#) is the [class \(or module\)](#) on which the method was called.
- In a block associated with a call to `instance_eval` [or](#) `instance_exec`, [self](#) is the object on which the method was called.

[self](#) automatically receives message that don't have an explicit receiver:

```
class String
  def upcase_and_reverse
    upcase.reverse
  end
end
```

In this method definition, the message `upcase` goes to [self](#), which is whatever string calls the method.

## [super](#)

Called from a method, searches along the method lookup path (the classes [and](#) modules available to the current object) [for](#) the [next](#) method of the same name as the one being executed. Such method, [if](#) present, may be defined [in](#) the superclass of the object's [class](#), but may also be defined [in](#) the superclass's superclass [or](#) any [class](#) on the upward path, as well as any [module](#) mixed [in](#) to any of those classes.

```

module Vehicular
  def move_forward(n)
    @position += n
  end
end

class Vehicle
  include Vehicular # Adds Vehicular to the lookup path
end

class Car < Vehicle
  def move_forward(n)
    puts "Vrooom!"
    super           # Calls Vehicular#move_forward
  end
end

```

Called with no arguments [and](#) no empty argument list, [super](#) calls the appropriate method with the same arguments, [and](#) the same code block, as those used to call the current method. Called with an argument list [or](#) arguments, it calls the appropriate methods with exactly the specified arguments (including none, [in](#) the [case](#) of an empty argument list indicated by empty parentheses).

### [then](#)

Optional component of conditional statements ([if](#), [unless](#), [when](#)). Never mandatory, but allows [for](#) one-line conditionals without semi-colons. The following two statements are equivalent:

```

if a > b; puts "a wins!" end
if a > b then puts "a wins!" end

```

See [if](#) [for](#) more examples.

### [true](#)

The sole instance of the special [class](#) TrueClass. [true](#) encapsulates Boolean truth; however, [<emph>all</emph>](#) objects [in](#) Ruby are [true in](#) the Boolean sense (informally, they cause an [if](#) test to succeed), with the exceptions of [false](#) and [nil](#).

Because Ruby regards most objects ([and](#) therefore most expressions) as "[true](#)", it is [not](#) always necessary to [return true](#) from a method to force a condition to succeed. However, it's good practice to [do](#) so, as it makes the intention clear.

### [undef](#)

Undefines a given method, [for](#) the [class or module in](#) which it's called. If the method is defined higher up [in](#) the lookup path (such as by a superclass), it can still be called by instances classes higher up.

```

class C
  def m
    "Hi"
  end
end
class D < C
end
class E < D
end

class D
  undef m
end

```

```
C.new.m    # Hi
D.new.m    # error
E.new.m    # error
```

Note that the argument to [undef](#) is a method name, [not](#) a symbol [or](#) string.

## [unless](#)

The negative equivalent of [if](#).

```
unless y.score > 10
  puts "Sorry; you needed 10 points to win."
end
```

See [if](#).

## [until](#)

The inverse of [while](#): executes code [until](#) a given condition is [true](#), i.e., [while](#) it is [not true](#). The semantics are the same as those of [while](#); see [while](#).

## [when](#)

See [case](#).

## [while](#)

[while](#) takes a condition argument, [and](#) executes the code that follows (up to a matching [end](#) delimiter) [while](#) the condition is [true](#).

```
i = 0
while i < 10
  i += 1
end
```

The value of the whole [while](#) statement is the value of the last expression evaluated the last time through the code. If the code is [not](#) executed (because the condition is [false](#) at the beginning of the operation), the [while](#) statement evaluates to [nil](#).

[while](#) can also appear [in](#) modifier position, either [in](#) a single-line statement [or in](#) a multi-line statement using a [begin/end](#) block. In the one-line [case](#):

```
i = 0 i += 1 while i < 10
```

the leading code is [not](#) executed at all [if](#) the condition is [false](#) at the start. However, [in](#) the "[begin](#)"-block [case](#):

```
i = 0
begin
  i += 1
  puts i
end while i < 10
```

the block will be executed at least once, before the condition is tested the first time.

## [yield](#)

Called from inside a method body, yields control to the code block ([if](#) any) supplied as part of the method call. If no code block has been supplied, calling [yield](#) raises an exception.

yield can take an argument; any values thus yielded are bound to the block's parameters. The value of a call to yield is the value of the executed code block.

[\[Validate\]](#)