Calling Methods

Calling a method sends a message to an object so it can perform some work.

In ruby you send a message to an object like this:

```
my_method()
```

Note that the parenthesis are optional:

```
my_method
```

Except when there is difference between using and omitting parentheses, this document uses parenthesis when arguments are present to avoid confusion.

This section only covers calling methods. See also the syntax documentation on defining methods

Receiver

self is the default receiver. If you don't specify any receiver self will be used. To specify a receiver use .:

```
my_object.my_method
```

This sends the my_method message to my_object. Any object can be a receiver but depending on the method's visibility sending a message may raise a NoMethodError.

You may also use :: to designate a receiver, but this is rarely used due to the potential for confusion with :: for namespaces.

Arguments

There are three types of arguments when sending a message, the positional arguments, keyword (or named) arguments and the block argument. Each message sent may use one, two or all types of arguments, but the arguments must be supplied in this order.

All arguments in ruby are passed by reference and are not lazily evaluated.

Each argument is separated by a ,:

```
my_method(1, '2', :three)
```

Arguments may be an expression, a hash argument:

```
'key' => value
```

or a keyword argument:

```
key: value
```

Hash and keyword arguments must be contiguous and must appear after all positional arguments, but may be mixed:

```
my_method('a' => 1, b: 2, 'c' => 3)
```

Positional Arguments

The positional arguments for the message follow the method name:

```
my_method(argument1, argument2)
```

In many cases parenthesis are not necessary when sending a message:

```
my_method argument1, argument2
```

However, parenthesis are necessary to avoid ambiguity. This will raise a SyntaxError because ruby does not know which method argument3 should be sent to:

```
method_one argument1, method_two argument2, argument3
```

If the method definition has a *argument extra positional arguments will be assigned to argument in the method as an Array.

If the method definition doesn't include keyword arguments the keyword or hash-type arguments are assigned as a single hash to the last argument:

```
def my_method(options)
  p options
end
```

```
my_method('a' => 1, b: 2) # prints: {'a'=>1, :b=>2}
```

If too many positional arguments are given an ArgumentError is raised.

Default Positional Arguments

When the method defines default arguments you do not need to supply all the arguments to the method. Ruby will fill in the missing arguments in-order.

First we'll cover the simple case where the default arguments appear on the right. Consider this method:

```
def my_method(a, b, c = 3, d = 4)
  p [a, b, c, d]
end
```

Here c and d have default values which ruby will apply for you. If you send only two arguments to this method:

```
my_method(1, 2)
```

You will see ruby print [1, 2, 3, 4].

If you send three arguments:

```
my_method(1, 2, 5)
```

You will see ruby print [1, 2, 5, 4]

Ruby fills in the missing arguments from left to right.

Ruby allows default values to appear in the middle of positional arguments. Consider this more complicated method:

```
def my_method(a, b = 2, c = 3, d)
  p [a, b, c, d]
end
```

Here b and c have default values. If you send only two arguments to this method:

```
my_method(1, 4)
```

You will see ruby print [1, 2, 3, 4].

If you send three arguments:

```
my_method(1, 5, 6)
```

You will see ruby print [1, 5, 3, 6].

Describing this in words gets complicated and confusing. I'll describe it in variables and values instead.

First 1 is assigned to a, then 6 is assigned to d. This leaves only the arguments with default values. Since 5 has not been assigned to a value yet, it is given to b and c uses its default value of 3.

Keyword Arguments

Keyword arguments follow any positional arguments and are separated by commas like positional arguments:

```
my_method(positional1, keyword1: value1, keyword2: value2)
```

Any keyword arguments not given will use the default value from the method definition. If a keyword argument is given that the method did not list an ArgumentError will be raised.

Block Argument

The block argument sends a closure from the calling scope to the method.

The block argument is always last when sending a message to a method. A block is sent to a method using do ... end or { ... }:

```
my_method do
  # ...
end
```

or:

```
my_method {
  # ...
}
```

do end has lower precedence than { } so:

```
method_1 method_2 {
  # ...
}
```

Sends the block to method_2 while:

```
method_1 method_2 do
  # ...
end
```

Sends the block to method_1. Note that in the first case if parentheses are used the block is sent to method_1.

A block will accept arguments from the method it was sent to. Arguments are defined similar to the way a method defines arguments. The block's arguments go in | ... | following the opening do or {:

```
my_method do |argument1, argument2|
  # ...
end
```

Block Local Arguments

You may also declare block-local arguments to a block using ; in the block arguments list. Assigning to a block-local argument will not override local arguments outside the block in the caller's scope:

```
def my_method
  yield self
end

place = "world"

my_method do |obj; place|
  place = "block"
  puts "hello #{obj} this is #{place}"
end

puts "place is: #{place}"
```

This prints:

```
hello main this is block
place is world
```

So the place variable in the block is not the same place variable as outside the block. Removing ; place from the block arguments gives this result:

```
hello main this is block
place is block
```

Array to Arguments Conversion

Given the following method:

```
def my_method(argument1, argument2, argument3)
end
```

You can turn an Array into an argument list with * (or splat) operator:

```
arguments = [1, 2, 3]
my_method(*arguments)
```

or:

```
arguments = [2, 3]
my_method(1, *arguments)
```

Both are equivalent to:

```
my_method(1, 2, 3)
```

If the method accepts keyword arguments the splat operator will convert a hash at the end of the array into keyword arguments:

```
def my_method(a, b, c: 3)
end

arguments = [1, 2, { c: 4 }]
my_method(*arguments)
```

You may also use the ** (described next) to convert a Hash into keyword arguments.

If the number of objects in the Array do not match the number of arguments for the method an

ArgumentError will be raised.

If the splat operator comes first in the call, parentheses must be used to avoid a warning.

Hash to Keyword Arguments Conversion

Given the following method:

```
def my_method(first: 1, second: 2, third: 3)
end
```

You can turn a Hash into keyword arguments with the ** operator:

```
arguments = { first: 3, second: 4, third: 5 }
my_method(**arguments)
```

or:

```
arguments = { first: 3, second: 4 }
my_method(third: 5, **arguments)
```

Both are equivalent to:

```
my_method(first: 3, second: 4, third: 5)
```

If the method definition uses ** to gather arbitrary keyword arguments they will not be gathered by *:

```
def my_method(*a, **kw)
  p arguments: a, keywords: kw
end

my_method(1, 2, '3' => 4, five: 6)
```

Prints:

```
{:arguments=>[1, 2, {"3"=>4}], :keywords=>{:five=>6}}
```

Unlike the splat operator described above the ** operator has no commonly recognized name.

Proc to Block Conversion

Given a method that use a block:

```ruby
def my_method
  yield self
end
```

You can convert a proc or lambda to a block argument with the & operator:

```ruby
argument = proc { |a| puts "#{a.inspect} was yielded" }

my_method(&argument)
```

If the splat operator comes first in the call, parenthesis must be used to avoid a warning.

Unlike the splat operator described above the & operator has no commonly recognized name.

Method Lookup

When you send a message Ruby looks up the method that matches the name of the message for the receiver. Methods are stored in classes and modules so method lookup walks these, not the objects themselves.

Here is the order of method lookup for the receiver's class or module R:

The prepended modules of R in reverse order

For a matching method in R

The included modules of R in reverse order

If R is a class with a superclass, this is repeated with R's superclass until a method is found.

Once a match is found method lookup stops.

If no match is found this repeats from the beginning, but looking for method_missing. The default method_missing is BasicObject#method_missing which raises a NameError when invoked.

If refinements (an experimental feature) are active the method lookup changes. See the refinements documentation for details.