Methods

Methods implement the functionality of your program. Here is a simple method definition:

```
def one_plus_one
  1 + 1
end
```

A method definition consists of the def keyword, a method name, the body of the method, return value and the end keyword. When called the method will execute the body of the method. This method returns 2.

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

Method Names

Method names may be one of the operators or must start a letter or a character with the eight bit set. It may contain letters, numbers, an _ (underscore or low line) or a character with the eight bit set. The convention is to use underscores to separate words in a multiword method name:

```
def method_name
  puts "use underscores to separate words"
end
```

Ruby programs must be written in a US-ASCII-compatible character set such as UTF-8, ISO-8859-1 etc. In such character sets if the eight bit is set it indicates an extended character. Ruby allows method names and other identifiers to contain such characters. Ruby programs cannot contain some characters like ASCII NUL x00.

The following are the examples of valid ruby methods:

```
def hello
  "hello"
end

def こんにちは
  puts "means hello in Japanese"
end
```

Typically method names are US-ASCII compatible since the keys to type them exist on all keyboards.

Method names may end with a ! (bang or exclamation mark), a ? (question mark) or = equals sign.

The bang methods(! at the end of method name) are called and executed just like any other method. However, by convention, a method with an exclamation point or bang is considered dangerous. In ruby core library the dangerous method implies that when a method ends with a bang(!), it indicates that unlike its non-bang equivalent, permanently modifies its receiver. Almost always, Ruby core library will have a non-bang counterpart(method name which does NOT end with !) of every bang method (method name which does end with !) that has does not modify the receiver. This convention is typically true for ruby core libary but may/may not hold true for other ruby libraries.

Methods that end with a question mark by convention return boolean. But they may not always return just true or false. Often they will may return an object to indicate a true value (or "truthy" value).

Methods that end with an equals sign indicate an assignment method. For assignment methods the return value is ignored, the arguments are returned instead.

These are method names for the various ruby operators. Each of these operators accept only one argument. Following the operator is the typical use or name of the operator. Creating an alternate meaning for the operator may lead to confusion as the user expects plus to add things, minus to subtract things, etc. Additionally, you cannot alter the precedence of the operators.

```
+
# add

-
# subtract

*
# multiply

**
# power

/
# divide

%
# modulus division, String#%

&
# AND

^
# XOR (exclusive OR)

>>
# right-shift

<<
# left-shift, append
```

```
==
# equal

!=
# not equal

===
# case equality. See Object#===

=~
# pattern match. (Not just for regular expressions)

!~
# does not match

<=>
# comparison aka spaceship operator. See Comparable

<
# less-than

<=
# less-than or equal

>
# greater-than

>=
# greater-than or equal
```

To define unary methods minus, plus, tilde and not (!) follow the operator with an @ as in +@ or !@:

```ruby
class C
  def -@
    puts "you inverted this object"
  end
end

obj = C.new

-obj # prints "you inverted this object"
```

Unary methods accept zero arguments.

Additionally, methods for element reference and assignment may be defined: [] and []= respectively. Both can take one or more arguments, and element reference can take none.

```ruby
class C
```

```
  def [](a, b)
    puts a + b
  end

  def []=(a, b, c)
    puts a * b + c
  end
end

obj = C.new

obj[2, 3]     # prints "5"
obj[2, 3] = 4 # prints "10"
```

Return Values

By default, a method returns the last expression that was evaluated in the body of the method. In the example above, the last (and only) expression evaluated was the simple sum 1 + 1. The return keyword can be used to make it explicit that a method returns a value.

```
def one_plus_one
  return 1 + 1
end
```

It can also be used to make a method return before the last expression is evaluated.

```
def two_plus_two
  return 2 + 2
  1 + 1  # this expression is never evaluated
end
```

Note that for assignment methods the return value will always be ignored. Instead the argument will be returned:

```
def a=(value)
  return 1 + value
end

p(a = 5) # prints 5
```

Scope

The standard syntax to define a method:

```
def my_method
  # ...
end
```

adds the method to a class. You can define an instance method on a specific class with the class keyword:

```
class C
  def my_method
    # ...
  end
end
```

A method may be defined on another object. You may define a “class method” (a method that is defined on the class, not an instance of the class) like this:

```
class C
  def self.my_method
    # ...
  end
end
```

However, this is simply a special case of a greater syntactical power in Ruby, the ability to add methods to any object. Classes are objects, so adding class methods is simply adding methods to the Class object.

The syntax for adding a method to an object is as follows:

```
greeting = "Hello"

def greeting.broaden
  self + ", world!"
end

greeting.broaden # returns "Hello, world!"
```

self is a keyword referring to the current object under consideration by the compiler, which might make the use of self in defining a class method above a little clearer. Indeed, the example of adding a hello method to the class String can be rewritten thus:

```
def String.hello
  "Hello, world!"
end
```

A method defined like this is called a “singleton method”. broaden will only exist on the string instance greeting. Other strings will not have broaden.

Overriding

When Ruby encounters the def keyword, it doesn’t consider it an error if the method already exists: it simply redefines it. This is called overriding. Rather like extending core classes, this is a potentially dangerous ability, and should be used sparingly because it can cause unexpected results. For example, consider this irb session:

```
>> "43".to_i
=> 43
>> class String
>>   def to_i
>>     42
>>   end
>> end
=> nil
>> "43".to_i
=> 42
```

This will effectively sabotage any code which makes use of the method String#to_i to parse numbers from strings.

Arguments

A method may accept arguments. The argument list follows the method name:

```ruby
def add_one(value)
  value + 1
end
```

When called, the user of the add_one method must provide an argument. The argument is a local variable in the method body. The method will then add one to this argument and return the value. If given 1 this method will return 2.

The parentheses around the arguments are optional:

```ruby
def add_one value
  value + 1
end
```

Multiple arguments are separated by a comma:

```
def add_values(a, b)
  a + b
end
```

When called, the arguments must be provided in the exact order. In other words, the arguments are positional.

Default Values

Arguments may have default values:

```
def add_values(a, b = 1)
  a + b
end
```

The default value does not need to appear first, but arguments with defaults must be grouped together. This is ok:

```
def add_values(a = 1, b = 2, c)
  a + b + c
end
```

This will raise a SyntaxError:

```
def add_values(a = 1, b, c = 1)
  a + b + c
end
```

Array Decomposition

You can decompose (unpack or extract values from) an Array using extra parentheses in the arguments:

```
def my_method((a, b))
  p a: a, b: b
end

my_method([1, 2])
```

This prints:

```
{:a=>1, :b=>2}
```

If the argument has extra elements in the Array they will be ignored:

```
def my_method((a, b))
  p a: a, b: b
end

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

This has the same output as above.

You can use a * to collect the remaining arguments. This splits an Array into a first element and the rest:

```
def my_method((a, *b))
  p a: a, b: b
end

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

This prints:

```
{:a=>1, :b=>[2, 3]}
```

The argument will be decomposed if it responds to to_ary. You should only define to_ary if you can use your object in place of an Array.

Use of the inner parentheses only uses one of the sent arguments. If the argument is not an Array it will be assigned to the first argument in the decomposition and the remaining arguments in the decomposition will be nil:

```
def my_method(a, (b, c), d)
  p a: a, b: b, c: c, d: d
end

my_method(1, 2, 3)
```

This prints:

```
{:a=>1, :b=>2, :c=>nil, :d=>3}
```

You can nest decomposition arbitrarily:

```
def my_method(((a, b), c))
  # ...
end
```

Array/Hash Argument

Prefixing an argument with * causes any remaining arguments to be converted to an Array:

```
def gather_arguments(*arguments)
  p arguments
end

gather_arguments 1, 2, 3 # prints [1, 2, 3]
```

The array argument must be the last positional argument, it must appear before any keyword arguments.

The array argument will capture a Hash as the last entry if a hash was sent by the caller after all positional arguments.

```
gather_arguments 1, a: 2 # prints [1, {:a=>2}]
```

However, this only occurs if the method does not declare any keyword arguments.

```
def gather_arguments_keyword(*positional, keyword: nil)
 p positional: positional, keyword: keyword
end

gather_arguments_keyword 1, 2, three: 3
#=> raises: unknown keyword: three (ArgumentError)
```

Also, note that a bare * can be used to ignore arguments:

```
def ignore_arguments(*)
end
```

Keyword Arguments

Keyword arguments are similar to positional arguments with default values:

```ruby
def add_values(first: 1, second: 2)
  first + second
end
```

Arbitrary keyword arguments will be accepted with **:

```ruby
def gather_arguments(first: nil, **rest)
  p first, rest
end

gather_arguments first: 1, second: 2, third: 3
# prints 1 then {:second=>2, :third=>3}
```

When calling a method with keyword arguments the arguments may appear in any order. If an unknown keyword argument is sent by the caller an ArgumentError is raised.

When mixing keyword arguments and positional arguments, all positional arguments must appear before any keyword arguments.

Block Argument

The block argument is indicated by & and must come last:

```ruby
def my_method(&my_block)
  my_block.call(self)
end
```

Most frequently the block argument is used to pass a block to another method:

```ruby
def each_item(&block)
  @items.each(&block)
end
```

If you are only going to call the block and will not otherwise manipulate it or send it to another method using yield without an explicit block parameter is preferred. This method is equivalent to the first method in this section:

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

There is also a performance benefit to using yield over a calling a block parameter. When a block argument is assigned to a variable a Proc object is created which holds the block. When using yield this Proc object is not created.

If you only need to use the block sometimes you can use Proc.new to create a proc from the block that was passed to your method. See Proc.new for further details.

Exception Handling

Methods have an implied exception handling block so you do not need to use begin or end to handle exceptions. This:

```
def my_method
  begin
    # code that may raise an exception
  rescue
    # handle exception
  end
end
```

May be written as:

```
def my_method
  # code that may raise an exception
rescue
  # handle exception
end
```

If you wish to rescue an exception for only part of your method use begin and end. For more details see the page on exception handling.