Chapter 8

More About Methods

So far in this book, we've been defining and using methods without much thought. Now it's time to get into the details.

Defining a Method

As we've seen, a method is defined using the keyword def. Method names should begin with a lowercase letter or underscore,[1] followed by letters, digits, and underscores.

A method name may end with one of ?, !, or =. Methods that return a boolean result (so-called predicate methods) are often named with a trailing ?:

```
1.even?                 # =>   false
2.even?                 # =>   true
1.instance_of?(Fixnum)  # =>   true
```

Methods that are "dangerous," or that modify their receiver, may be named with a trailing exclamation mark, !. These are sometimes called *bang methods*. For instance, String provides both chop and chop! methods. The first one returns a modified string; the second modifies the receiver in place.

Methods that can appear on the left side of an assignment (a feature we discussed on page 55) end with an equals sign (=).

?, !, and = are the only "weird" characters allowed as method name suffixes.

Now that we've specified a name for our new method, we may need to declare some parameters. These are simply a list of local variable names in parentheses. (The parentheses around a method's arguments are optional; our convention is to use them when a method has arguments and omit them when it doesn't.)

1. You won't get an immediate error if you start a method name with an uppercase letter, but when Ruby sees you calling the method, it might guess that it is a constant, not a method invocation, and as a result it may parse the call incorrectly. By convention, methods names starting with an uppercase letter are used for type conversion. The Integer method, for example, converts its parameter to an integer.

```
def my_new_method(arg1, arg2, arg3)     # 3 arguments
  # Code for the method would go here
end
def my_other_new_method                  # No arguments
  # Code for the method would go here
end
```

Ruby lets you specify default values for a method's arguments—values that will be used if the caller doesn't pass them explicitly. You do this using an equals sign (=) followed by a Ruby expression. That expression can include references to previous arguments in the list:

```
def cool_dude(arg1="Miles", arg2="Coltrane", arg3="Roach")
  "#{arg1}, #{arg2}, #{arg3}."
end

cool_dude                               # =>   "Miles, Coltrane, Roach."
cool_dude("Bart")                       # =>   "Bart, Coltrane, Roach."
cool_dude("Bart", "Elwood")             # =>   "Bart, Elwood, Roach."
cool_dude("Bart", "Elwood", "Linus")    # =>   "Bart, Elwood, Linus."
```

Here's an example where the default argument references a previous argument:

```
def surround(word, pad_width=word.length/2)
  "[" * pad_width  + word +  "]" * pad_width
end

surround("elephant")   # =>   "[[[[elephant]]]]"
surround("fox")        # =>   "[fox]"
surround("fox", 10)    # =>   "[[[[[[[[[[fox]]]]]]]]]]"
```

The body of a method contains normal Ruby expressions. The return value of a method is the value of the last expression executed or the result of an explicit return expression.

Variable-Length Argument Lists

But what if you want to pass in a variable number of arguments or want to capture multiple arguments into a single parameter? Placing an asterisk before the name of the parameter after the "normal" parameters lets you do just that. This is sometimes called *splatting an argument* (presumably because the asterisk looks somewhat like a bug after hitting the windscreen of a fast moving car).

```
def varargs(arg1, *rest)
  "arg1=#{arg1}.  rest=#{rest.inspect}"
end

varargs("one")                  # =>   arg1=one.  rest=[]
varargs("one", "two")           # =>   arg1=one.  rest=[two]
varargs "one", "two", "three"   # =>   arg1=one.  rest=[two, three]
```

In this example, the first argument is assigned to the first method parameter as usual. However, the next parameter is prefixed with an asterisk, so all the remaining arguments are bundled into a new Array, which is then assigned to that parameter.

Folks sometimes use a splat to specify arguments that are not used by the method (but that are perhaps used by the corresponding method in a superclass. (Note that in this example we call super with no parameters. This is a special case that means "invoke this method in the superclass, passing it all the parameters that were given to the original method.")

```
class Child < Parent
  def do_something(*not_used)
    # our processing
    super
  end
end
```

In this case, you can also leave off the name of the parameter and just write an asterisk:

```
class Child < Parent
  def do_something(*)
    # our processing
    super
  end
end
```

1.9 In Ruby 1.9, you can put the splat argument anywhere in a method's parameter list, allowing you to write this:

```
def split_apart(first, *splat, last)
  puts "First: #{first.inspect}, splat: #{splat.inspect}, " +
       "last: #{last.inspect}"
end
split_apart(1,2)
split_apart(1,2,3)
split_apart(1,2,3,4)
```

produces:

```
First: 1, splat: [], last: 2
First: 1, splat: [2], last: 3
First: 1, splat: [2, 3], last: 4
```

If you cared only about the first and last parameters, you could define this method using this:

```
def split_apart(first, *, last)
  # ...
end
```

You can have only one splat argument in a method—if you had two, it would be ambiguous. You also can't put arguments with default values after the splat argument. In all cases, the splat argument receives the values left over after assigning to the regular argument.

Methods and Blocks

As we discussed in the section on blocks and iterators beginning on page 74, when a method is called, it may be associated with a block.

Normally, you simply call the block from within the method using yield:

Download samples/tutmethods_10.rb

```
def double(p1)
  yield(p1*2)
end

double(3) {|val| "I got #{val}" }            # =>   "I got 6"
double("tom") {|val| "Then I got #{val}" }   # =>   "Then I got tomtom"
```

However, if the last parameter in a method definition is prefixed with an ampersand, any associated block is converted to a Proc object, and that object is assigned to the parameter. This allows you to store the block for use later.

Download samples/tutmethods_11.rb

```
class TaxCalculator
  def initialize(name, &block)
    @name, @block = name, block
  end
  def get_tax(amount)
    "#@name on #{amount} = #{ @block.call(amount) }"
  end
end

tc = TaxCalculator.new("Sales tax") {|amt| amt * 0.075 }

tc.get_tax(100)   # =>   "Sales tax on 100 = 7.5"
tc.get_tax(250)   # =>   "Sales tax on 250 = 18.75"
```

Calling a Method

You call a method by optionally specifying a receiver, giving the name of the method, and optionally passing some parameters and an optional block. Here's a code fragment that shows us calling a method with a receiver, a parameter, and a block:

```
connection.download_mp3("jitterbug") {|p| show_progress(p) }
```

In this example, the object connection is the receiver, download_mp3 is the name of the method, the string "jitterbug" is the parameter, and the stuff between the braces is the associated block. During this method call, Ruby first sets self to the receiver and then invokes the method in that object: For class and module methods, the receiver will be the class or module name.

```
File.size("testfile")   # =>   66
Math.sin(Math::PI/4)    # =>   0.707106781186547
```

If you omit the receiver, it defaults to self, the current object.

```
class InvoiceWriter
   def initialize(order)
     @order = order
   end
```

```
    def write_on(output)
      write_header_on(output)     # called on current object.
      write_body_on(output)       # self is not changed, as
      write_totals_on(output)     # there is no receiver
    end
    def write_header_on(output)
      # ...
    end
    def write_body_on(output)
      # ...
    end
    def write_totals_on(output)
      # ...
    end
end
writer = InvoiceWriter.new(my_order)
writer.write_on(STDOUT)
```

This defaulting mechanism is how Ruby implements private methods. Private methods may *not* be called with a receiver, so they must be methods available in the current object. In the previous example, we'd probably want to make the helper methods private, because they shouldn't be called from outside the InvoiceWriter class:

```
class InvoiceWriter
    def initialize(order)
      @order = order
    end
    def write_on(output)
      write_header_on(output)
      write_body_on(output)
      write_totals_on(output)
    end
private
    def write_header_on(output)
      # ...
    end
    def write_body_on(output)
      # ...
    end
    def write_totals_on(output)
      # ...
    end
end
```

Passing Parameters to a Method

Any parameters follow the method name. If no ambiguity exists, you can omit the parentheses around the argument list when calling a method.[2] However, except in the simplest cases we don't recommend this—some subtle problems can trip you up.[3] Our rule is simple: if you have any doubt, use parentheses.

```
a = obj.hash    # Same as
a = obj.hash()  # this.
obj.some_method "Arg1", arg2, arg3   # Same thing as
obj.some_method("Arg1", arg2, arg3)  # with parentheses.
```

Older Ruby versions compounded the problem by allowing you to put spaces between the method name and the opening parenthesis. This made it hard to parse: is the parenthesis the start of the parameters or the start of an expression? As of Ruby 1.8 you get a warning if you put a space between a method name and an open parenthesis.

Method Return Values

Every called method returns a value (although there's no rule that says you have to use that value). The value of a method is the value of the last statement executed by the method:

```
def meth_one
  "one"
end
meth_one   # =>   "one"

def meth_two(arg)
  case
  when arg > 0  then "positive"
  when arg < 0  then "negative"
  else               "zero"
  end
end

meth_two(23)   # =>   "positive"
meth_two(0)    # =>   "zero"
```

Ruby has a return statement, which exits from the currently executing method. The value of a return is the value of its argument(s). It is idiomatic Ruby to omit the return if it isn't needed, as shown by the previous two examples.

---

2. Other Ruby documentation sometimes calls these method calls without parentheses *commands*.

3. In particular, you *must* use parentheses on a method call that is itself a parameter to another method call (unless it is the last parameter).

This next example uses return to exit from a loop inside the method:

```
def meth_three
 100.times do |num|
   square = num*num
   return num, square if square > 1000
 end
end
meth_three   # =>   [32, 1024]
```

As the last case illustrates, if you give return multiple parameters, the method returns them in an array. You can use parallel assignment to collect this return value:

```
num, square = meth_three
num      # =>   32
square   # =>   1024
```

Splat! Expanding Collections in Method Calls

Earlier we saw that if you put an asterisk in front of a parameter in a method definition, multiple arguments in the call to the method will be bundled into an array. Well, the same thing works in reverse.

1.9

When you call a method, you can convert any collection or enumerable object into its constituent elements and pass those elements as individual parameters to the method. Do this by prefixing array arguments with an asterisk:

```
def five(a, b, c, d, e)
  "I was passed #{a} #{b} #{c} #{d} #{e}"
end

five(1, 2, 3, 4, 5 )         # =>   "I was passed 1 2 3 4 5"
five(1, 2, 3, *['a', 'b'])   # =>   "I was passed 1 2 3 a b"
five(*['a', 'b'], 1, 2, 3)   # =>   "I was passed a b 1 2 3"
five(*(10..14))              # =>   "I was passed 10 11 12 13 14"
five(*[1,2], 3, *(4..5))     # =>   "I was passed 1 2 3 4 5"
```

1.9

As of Ruby 1.9, splat arguments can appear anywhere in the parameter list, and you can intermix splat and regular arguments.

Making Blocks More Dynamic

We've already seen how to associate a block with a method call:

```
for_each_bone(aardvark) do |bone|
  # ...
end
```

Normally, this is perfectly good enough—you associate a fixed block of code with a method in the same way you'd have a chunk of code after an if or while statement.

Sometimes, however, you'd like to be more flexible. For example, we may be teaching math skills.[4] The student could ask for an *n*-plus table or an *n*-times table. If the student asked for a 2-times table, we'd output 2, 4, 6, 8, and so on. (This code does not check its inputs for errors.)

Download **samples/tutmethods_23.rb**

```
print "(t)imes or (p)lus: "
operator = gets
print "number: "
number = Integer(gets)
if operator =~ /^t/
  puts((1..10).collect {|n| n*number }.join(", "))
else
  puts((1..10).collect {|n| n+number }.join(", "))
end
```

produces:

```
(t)imes or (p)lus: t
number: 2
2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

This works, but it's ugly, with virtually identical code on each branch of the if statement. It would be nice if we could factor out the block that does the calculation:

Download **samples/tutmethods_24.rb**

```
print "(t)imes or (p)lus: "
operator = gets
print "number: "
number = Integer(gets)
if operator =~ /^t/
  calc = lambda {|n| n*number }
else
  calc = lambda {|n| n+number }
end
puts((1..10).collect(&calc).join(", "))
```

produces:

```
(t)imes or (p)lus: t
number: 2
2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

If the last argument to a method is preceded by an ampersand, Ruby assumes that it is a Proc object. It removes it from the parameter list, converts the Proc object into a block, and associates it with the method.

4. Of course, Andy and Dave would have to *learn* math skills first. Conrad Schneiker reminded us that there are three kinds of people: those who can count and those who can't.

Collecting Hash Arguments

Some languages feature *keyword arguments*. Instead of passing a specific number of arguments in a given order, you can invoke the method with the names of the arguments, each with a corresponding value, in any order. Ruby 1.9 does not have keyword arguments, although they might appear in Ruby 2.0. In the meantime, people are using hashes as a way of achieving the same effect. For example, we could consider adding a search facility to an MP3 playlist:

```
class SongList
  def search(name, params)
    # ...
  end
end
list.search(:titles,
            { :genre              => "jazz",
              :duration_less_than => 270
            })
```

The first parameter tells the search what to return. The second parameter is a hash literal containing search parameters. (Note how we used symbols as the keys for this options hash. This has become idiomatic in Ruby libraries and frameworks.) The use of a hash means we can simulate keywords: look for songs with a genre of "jazz" and a duration less than $4\frac{1}{2}$ minutes.

However, this approach is slightly clunky, and that set of braces could easily be mistaken for a block associated with the method. So, Ruby has a shortcut. You can place *key* => *value* pairs in an argument list, as long as they follow any normal arguments and precede any splat and block arguments. All these pairs will be collected into a single hash and passed as one argument to the method. No braces are needed.

```
list.search(:titles,
            :genre              => 'jazz',
            :duration_less_than => 270)
```

1.9 Option passing is one of the reasons for the new hash literal syntax in Ruby 1.9:

```
list.search(:titles, genre: 'jazz', duration_less_than: 270)
```

A well-written Ruby program will typically contain many methods, each quite small, so it's worth getting familiar with the options available when defining and using Ruby methods. At some point you'll probably want to read the section called *Method Arguments* on page 353 to see exactly how arguments in a method call get mapped to the method's formal parameters when you have combinations of default parameters and splat parameters.