More About Methods

Chapter 5

So far we've seen a number of different methods, puts and gets and so on (Pop Quiz: List all of the methods we have seen so far! There are ten of them; the answer is below.), but we haven't really talked about what methods are. We know what they do, but we don't know what they are.

But really, that is what they are: things that do stuff. If objects (like strings, integers, and floats) are the nouns in the Ruby language, then methods are like the verbs. And, just like in English, you can't have a verb without a noun to do the verb. For example, ticking isn't something that just happens; a clock (or a watch or something) has to do it. In English we would say, "The clock ticks." In Ruby we would say clock.tick (assuming that clock was a Ruby object, of course). Programmers might say we were "calling clock's tick method," or that we "called tick on clock."

So, did you take the quiz? Good. Well, I'm sure you remembered the methods puts, gets, and chomp, since we just covered those. You probably also got our conversion methods, to_i, to_f, and to_s. However, did you get the other four? Why, it's none other than our old arithmetic buddies +, -, *, and /!

So as I was saying, just as every verb needs a noun, so every method needs an object. It's usually easy to tell which object is performing the method: it's what comes right before the dot, like in our clock.tick example, or in 101.to_s. Sometimes, however, it's not quite as obvious; like with the arithmetic methods. As it turns out, 5 + 5 is really just a shortcut way of writing 5.+ 5. For example:

```ruby
puts 'hello '.+ 'world'
puts (10.* 9).+ 9
```

```
hello world
99
```

It isn't very pretty, so we won't ever write it like that; however, it's important to understand what is really happening. (On older versions of Ruby, this code might also give a warning: warning: parenthesize argument(s) for future version. It would still run the code just fine, though.) This also gives us a deeper understanding of why we can do 'pig' * 5 but we can't do 5 * 'pig': 'pig' * 5 is telling 'pig' to do the multiplying, but 5 * 'pig' is telling 5 to do the multiplying. 'pig' knows how to make 5 copies of itself and add them all together; however, 5 will have a much more difficult time of making 'pig' copies of itself and adding them together.

And, of course, we still have puts and gets to explain. Where are their objects? In English, you can sometimes leave out the noun; for example, if a villain yells "Die!", the implicit noun is whoever he is yelling at. In Ruby, if I say puts 'to be or not to be', what I am really saying is self.puts 'to be or not to be'. So what is self? It's a special variable which points to whatever object you are in. We don't even know how to be in an object yet, but until we find out, we are always going to be in a big object which is... the whole program! And lucky for us, the program has a few methods of its own, like puts and gets. Watch this:

```
iCantBelieveIMadeAVariableNameThisLongJustToPointToA3 = 3
puts iCantBelieveIMadeAVariableNameThisLongJustToPointToA3
self.puts iCantBelieveIMadeAVariableNameThisLongJustToPointToA3
```

```
3
3
```

If you didn't entirely follow all of that, that's OK. The important thing to take away from all of this is that every method is being done by some object, even if it doesn't have a dot in front of it. If you understand that, then you're all set.

Fancy String Methods

Let's learn a few fun string methods. You don't have to memorize them all; you can just look up this page again if you forget them. I just want to show you a small part of what strings can do. In fact, I can't remember even half of the string methods myself—but that's fine, because there are great references on the internet with all of the string methods listed and explained. (I will show you where to find them at the end of this tutorial.) Really, I don't even want to know all the string methods; it's kind of like knowing every word in the dictionary. I can speak English just fine without knowing every word in the dictionary... and isn't that really the whole point of the dictionary? So you don't have to know what's in it?

So, our first string method is reverse, which gives a backwards version of a string:

```
var1 = 'stop'
var2 = 'stressed'
var3 = 'Can you pronounce this sentence backwards?'

puts var1.reverse
puts var2.reverse
puts var3.reverse
puts var1
puts var2
puts var3
```

```
pots
desserts
?sdrawkcab ecnetnes siht ecnuonorp uoy naC
stop
stressed
Can you pronounce this sentence backwards?
```

As you can see, reverse doesn't reverse the original string; it just makes a new backwards version of it. That's why var1 is still 'stop' even after we called reverse on var1.

Another string method is length, which tells us the number of characters (including spaces) in the string:

```
puts 'What is your full name?'
name = gets.chomp
puts 'Did you know there are ' + name.length +
     ' characters in your name, ' + name + '?'
```

```
What is your full name?
Christopher David Pine
#<TypeError: no implicit conversion of Fixnum into String>
```

Uh-oh! Something went wrong, and it looks like it happened sometime after the line name = gets.chomp... Do you see the problem? See if you can figure it out.

The problem is with length: it gives us a number, but we want a string. Easy enough, we'll just throw in a to_s (and cross our fingers):

```
puts 'What is your full name?'
name = gets.chomp
puts 'Did you know there are ' + name.length.to_s +
     ' characters in your name, ' + name + '?'
```

```
What is your full name?
Christopher David Pine
Did you know there are 22 characters in your name, Christopher David Pine?
```

No, I did not know that. Note: that's the number of characters in my name, not the number of letters (count 'em). I guess we could write a program which asks for your first, middle, and last names individually, and then adds those lengths together... hey, why don't you do that! Go ahead, I'll wait.

Did you do it? Good! It's nice to program, isn't it? After a few more chapters, though, you'll be amazed at what you can do.

So, there are also a number of string methods which change the case (uppercase and lowercase) of your string. upcase changes every lowercase letter to uppercase, and downcase changes every uppercase letter to lowercase. swapcase switches the case of every letter in the string, and finally, capitalize is just like downcase, except that it switches the first character to uppercase (if it is a letter).

```
letters = 'aAbBcCdDeE'
puts letters.upcase
puts letters.downcase
puts letters.swapcase
puts letters.capitalize
puts ' a'.capitalize
puts letters
```

```
AABBCCDDEE
aabbccddee
AaBbCcDdEe
Aabbccddee
 a
aAbBcCdDeE
```

Pretty standard stuff. As you can see from the line puts ' a'.capitalize, the method capitalize only capitalizes the first character, not the first letter. Also, as we have seen before, throughout all of these method calls, letters remains unchanged. I don't mean to belabor the point, but it's important to understand. There are some methods which do change the associated object, but we haven't seen any yet, and we won't for some time.

The last of the fancy string methods we'll look at are for visual formatting. The first one, center, adds spaces to the beginning and end of the string to make it centered. However, just like you have to tell puts what you want it to print, and + what you want it to add, you have to tell center how wide you want your centered string to be. So if I wanted to center the lines of a poem, I would do it like this:

```
lineWidth = 50
puts(                'Old Mother Hubbard'.center(lineWidth))
puts(               'Sat in her cupboard'.center(lineWidth))
puts(         'Eating her curds an whey,'.center(lineWidth))
puts(          'When along came a spider'.center(lineWidth))
puts(         'Which sat down beside her'.center(lineWidth))
puts('And scared her poor shoe dog away.'.center(lineWidth))
```

```
                Old Mother Hubbard
               Sat in her cupboard
            Eating her curds an whey,
             When along came a spider
            Which sat down beside her
        And scared her poor shoe dog away.
```

Hmmm... I don't think that's how that nursery rhyme goes, but I'm too lazy to look it up. (Also, I wanted to line up the .center lineWidth part, so I put in those extra spaces before the strings. This is just because I think it is prettier that way. Programmers often have strong feelings about what is pretty in a program, and they often disagree about it. The more you program, the more you will come into your own style.) Speaking of being lazy, laziness isn't always a bad thing in programming. For example, see how I stored the width of the poem in the variable lineWidth? This was so that if I want to go back later and make the poem wider, I only have to change the very top line of the program, instead of every line which does centering. With a very long poem, this could save me a lot of time. That kind of laziness is really a virtue in programming.

So, about that centering... you may have noticed that it isn't quite as beautiful as what a word processor would have done. If you really want perfect centering (and maybe a nicer font), then you should just use a word processor! Ruby is a wonderful tool, but no tool is the right tool for every job.

The other two string formatting methods are ljust and rjust, which stand for left justify and right justify. They are similar to center, except that they pad the string with spaces on the right and left sides, respectively. Let's take a look at all three in action:

```
lineWidth = 40
str = '--> text <--'
puts str.ljust  lineWidth
puts str.center lineWidth
puts str.rjust  lineWidth
puts str.ljust(lineWidth/2) + str.rjust(lineWidth/2)
```

```
--> text <--
              --> text <--
                            --> text <--
--> text <--                --> text <--
```

A Few Things to Try

Write an Angry Boss program. It should rudely ask what you want. Whatever you answer, the Angry Boss should yell it back to you, and then fire you. For example, if you type in I want a raise., it should yell back WHADDAYA MEAN "I WANT A RAISE."?!? YOU'RE FIRED!! So here's something for you to do in order to play around more with center, ljust, and rjust: Write a program which will display a Table of Contents so that it looks like this:

```
                Table of Contents

Chapter 1:  Numbers                        page 1
Chapter 2:  Letters                       page 72
Chapter 3:  Variables                    page 118
```

Higher Math

(This section is totally optional. It assumes a fair degree of mathematical knowledge. If you aren't interested, you can go straight to Flow Control without any problems. However, a quick look at the section on Random Numbers might come in handy.)

There aren't nearly as many number methods as there are string methods (though I still don't know them all off the top of my head). Here, we'll look at the rest of the arithmetic methods, a random number generator, and the Math object, with its trigonometric and transcendental methods.

More Arithmetic

The other two arithmetic methods are ** (exponentiation) and % (modulus). So if you want to say "five squared" in Ruby, you would write it as 5 ** 2. You can also use floats for your exponent, so if you want the square root of 5, you could write 5 ** 0.5. The modulus method gives you the remainder after division by a number. So, for example, if I divide 7 by 3, I get 2 with a remainder of 1. Let's see it working in a program:

```
puts 5 ** 2
puts 5 * 0.5
puts 7 / 3
puts 7 % 3
puts 365 % 7
```

```
25
2.23606797749979
2
1
1
```

From that last line, we learn that a (non-leap) year has some number of weeks, plus one day. So if your birthday was on a Tuesday this year, it will be on a Wednesday next year. You can also use floats with the modulus method. Basically, it works the only sensible way it could... but I'll let you play around with that.

There's one last method to mention before we check out the random number generator: abs. It just takes the absolute value of the number:

```
puts((5 - 2).abs)
puts((2 - 5).abs)
```

```
3
3
```

Random Numbers

Ruby comes with a pretty nice random number generator. The method to get a randomly chosen number is rand. If you call rand just like that, you'll get a float greater than or equal to 0.0 and less than 1.0. If you give rand an integer (5 for example), it will give you an integer greater than or equal to 0 and less than 5 (so five possible numbers, from 0 to 4).

Let's see rand in action.

```
puts rand
puts rand
puts rand
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(1))
puts(rand(1))
puts(rand(1))
puts(rand(99999999999999999999999999999999999999999999999999999999999))
puts('The weatherman said there is a ' + rand(101).to_s + '% chance of rain,')
puts('but you can never trust a weatherman.')
```

```
0.17429261270690644
0.6038796470847551
0.5357456897902644
82
63
21
0
0
0
93717128025224525218060865578711367433735357080012377789690
The weatherman said there is a 9% chance of rain,
but you can never trust a weatherman.
```

Note that I used rand(101) to get back numbers from 0 to 100, and that rand(1) always gives back 0. Not understanding the range of possible return values is the biggest mistake I see people make with rand; even professional programmers; even in finished products you can buy at the store. I even had a CD player once which, if set on "Random Play," would play every song but the last one... (I wonder what would have happened if I had put in a CD with only one song on it?)

Sometimes you might want rand to return the same random numbers in the same sequence on two different runs of your program. (For example, once I was using randomly generated numbers to create a randomly generated world for a computer game. If I found a world that I really liked, perhaps I would want to play on it again, or send it to a friend.) In order to do this, you need to set the seed, which you can do with srand. Like this:

```
srand 1776
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts ''
srand 1776
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
```

```
24
35
36
58
```

```
70


24
35
36
58
70
```

It will do the same thing every time you seed it with the same number. If you want to get different numbers again (like what happens if you never use srand), then just call srand 0. This seeds it with a really weird number, using (among other things) the current time on your computer, down to the millisecond.

The Math Object

Finally, let's look at the Math object. We might as well jump right in:

```
puts(Math::PI)
puts(Math::E)
puts(Math.cos(Math::PI / 3))
puts(Math.tan(Math::PI / 4))
puts(Math.log(Math::E ** 2))
puts((1 + Math.sqrt(5)) / 2)
```

```
3.141592653589793
2.718281828459045
0.5000000000000001
0.9999999999999999
2.0
1.618033988749895
```

The first thing you noticed was probably the :: notation. Explaining the scope operator (which is what that is) is really beyond the, uh... scope of this tutorial. No pun intended. I swear. Suffice it to say, you can use Math::PI just like you would expect to.

As you can see, Math has all of the things you would expect a decent scientific calculator to have. And as always, the floats are really close to being the right answers.

So now let's flow!