Extracted from:

Learn to Program

The Pragmatic Programmers

Learn to Program

Chris Pine

The Facets of Ruby Series

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragmaticprogrammer.com

Printed in the United States of America.

ISBN 0-9766940-4-2

Printed on acid-free paper.

Fourth printing, January 2009

Version: 2009-3-13

Chapter 6

More about Methods

So far we've seen a number of different methods—`puts`, `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.) However, we haven't really talked about what methods are.

I believe the technical definition is *methods are things that do stuff.* If objects (such as 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 (biological or otherwise) 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, and one that could `tick`). Programmers might say we were "calling `clock`'s `tick` method" or that we "called `tick` on `clock`,". (This goes a long way toward explaining why we aren't invited to many parties. We?? *They!* Why *they* aren't invited to many parties....)

Anyway, 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`. But did you get the other four? Yeah? No? Why, it's none other than our old arithmetic buddies: `+`, `-`, `*`, and `/`! (See, it's stuff like that, too. Arithmetic buddies?? It would have sounded *way* cooler if I had said it in Klingon....)

So as I was saying, just as every verb needs a noun, 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, though, it's not quite as obvious,

as with the arithmetic methods. As it turns out, `5 + 5` is really just a shortcut way of writing `5.+ 5`. For example:

```
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 my machine, that also gives me a *warning*:

```
warning: parenthesize argument(s) for future version
```

It still ran the code just fine, but it's telling me it's having trouble figuring out what I mean and to use more parentheses in the future.) 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 whomever he is yelling at. In Ruby, if I say `puts 'to be or not to be'`, the implicit object is whatever object you happen to be *in*. But we don't even know *how* to be in an object yet; we've always been inside a special object Ruby has created for us that represents the whole program. You can always see what object you are in by using the special variable `self`. Watch this:

```
puts self
```

```
main
```

If you didn't entirely follow all of that, that's OK. The important thing to get from all 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.

6.1 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 you can find great references on the Internet with all the string methods listed and explained. (I will show you where to find them in Chapter 15, *Beyond This Fine Book*, on page 151.) 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? You don't *have* to know what's in it.)

So, our first string method is `reverse`, which returns a reversed version of the string:

FF1 SPOILER ALERT! (From back when final used to mean something.)

```
var1 = 'stop'
var2 = 'deliver repaid desserts'
var3 = '....TCELES B HSUP  A magic spell?'

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

```
pots
stressed diaper reviled
?lleps cigam A  PUSH B SELECT....
stop
deliver repaid desserts
....TCELES B HSUP  A magic spell?
```

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

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'
puts 'in your name, ' + name + '?'
```

```
What is your full name?
Christopher David Pine
#<TypeError: cannot convert Fixnum into String>
```

Uh-oh! See? There it is! It's an easy mistake to make. Anyway, if you didn't know to be on the lookout for this error, you can still figure that the problem must have happened sometime after the line `name = gets.chomp`, since I was able to type my name. See whether you can figure it out.

You probably think that I made that mistake on purpose, since I'm obviously such a fabulous programmer that I'm writing a book on it...you were thinking that, right? Well...never mind.

The problem is with `length`: it gives us an integer, but we want a string. That's 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'
puts '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: 22 is the number of *characters* in my name, not the number of *letters* (count 'em). I guess we could write a program that 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? Right on.

Well, unless your name is Bjørn or Håvard, in which case you had some problems. Ruby is expecting only ASCII characters (basically the stuff you can type on an American keyboard—the *A* in ASCII stands for *American*). It is possible to use any character in any language, but it requires some extra work and is just more advanced than what we're going to cover.

So, a number of string methods can also 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 it switches the first character to uppercase (if it's 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
```

As you can see from the line `puts ' a'.capitalize`, the `capitalize` method capitalizes only 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. Some methods *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 do visual formatting. The first, `center`, adds spaces to the beginning and end of the string to make it centered. However, just like you have to tell the `puts` method what you want it to print, and the + method what you want it to add, you have to tell the `center` method 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:

I wanted to line up the `.center line_width` part, so I added those extra spaces before the strings. This is just because I think it is prettier that way. Programmers often have strong feelings about code aesthetics, and they often disagree about them. The more you program, the more you will come into your own style.

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

```
                Old Mother Hubbard
               Sat in her cupboard
           Eating her curds and whey,
            When along came a spider
            Who 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. Speaking of laziness, see how I stored the width of the poem in the variable `line_width`? This was so that if I want to go back later and make the poem wider, I have to change only the first line of the program, instead of every line that does centering. With a very long poem, this could save me a lot of time. That's the kind of laziness we want in our programs.

So, about that centering...you may have noticed that it isn't quite as beautiful as 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 we'll look at today 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 left and right sides, respectively. Let's take a look at all three in action:

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

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

6.2 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 like this:

```
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 that will display a table of contents so that it looks like this:

```
                Table of Contents

Chapter 1:  Getting Started                 page  1
Chapter 2:  Numbers                         page  9
Chapter 3:  Letters                         page 13
```

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help programmers stay on top of their game.

Visit Us Online

`pragmaticprogrammer.com/titles/fr_ltp`
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
`pragmaticprogrammer.com/updates`
Be notified when updates and new books become available.

Join the Community
`pragmaticprogrammer.com/community`
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
`pragmaticprogrammer.com/news`
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: `pragmaticprogrammer.com/titles/fr_ltp`.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	`www.pragmaticprogrammer.com/catalog`
Customer Service:	`orders@pragmaticprogrammer.com`
Non-English Versions:	`translations@pragmaticprogrammer.com`
Pragmatic Teaching:	`academic@pragmaticprogrammer.com`
Author Proposals:	`proposals@pragmaticprogrammer.com`