So far we’ve seen a number of different methods—puts, gets, and so on. (Pop quiz: List all 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 that “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. Arithmetic buddies? Seriously, Chris?)

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:

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

```ruby
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.

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:

```ruby
puts self
```

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 123.) 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.)

Our first string method is reverse, which returns a reversed version of the string:

```ruby
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

puts 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.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).

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

<table>
<thead>
<tr>
<th>AABBCDDDEE</th>
<th>aabbcdddee</th>
</tr>
</thead>
<tbody>
<tr>
<td>AabBcCdDDee</td>
<td>Aabbcdddee</td>
</tr>
<tr>
<td>aAbBcCdDDeE</td>
<td>aAbBcCdDDeE</td>
</tr>
</tbody>
</table>

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 you have to tell 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:

```
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))
```

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.

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 right and left 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 <--
 --> text <--
```
6.2 A Few Things to Try

- **Angry boss.** Write an angry boss program that rudely asks 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:

  WHADDA YA MEAN "I WANT A RAISE"?!!? YOU'RE FIRED!!

- **Table of contents.** 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

6.3 Higher Math

(This section is optional. Some of it assumes a fair degree of mathematical knowledge. If you aren’t interested, you can go straight to Chapter 7, Flow Control, on page 37, without any problems. However, a quick scan of this section 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.

6.4 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 you divide 7 by 3, you 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
From that last line, we learn that a (nonleap) 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.

I have one last method to mention before we check out the random number generator: abs. This method simply returns the absolute value of the number:

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

6.5 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 it an integer parameter (by calling `rand(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:

```ruby
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(99999999999999999999999999999999))
puts('The weatherman said there is a')
puts(rand(101).to_s + ' % chance of rain,')
puts('but you can never trust a weatherman. ')
```

```text
0.780420251671991
0.27612531216364
0.0633567492235
21
12
```
The weatherman said there is a 67% chance of rain, but you can never trust a weatherman.

Note that for the weatherman example I used rand(101) to get numbers from 0 to 100 and that rand(1) always returns 0. Not understanding the range of possible return values is the biggest mistake I see people make with rand, even professional programmers, and even in finished products you can buy at the store. I once had a CD player that 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, I used randomly generated numbers to generate the worlds in Civilization III. If I found a world that I really liked, I’d save it, run tests on it, and so on.) In order to do this, you need to set the seed, which you can do with srand:

```ruby
srand 1976
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts ''
srand 1976
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
```

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, passing in no parameter. This seeds it with a really weird
number, using (among other things) the current time on your computer, down to the millisecond.

6.6 **The Math Object**

Finally, let’s look at the `Math` object. They say a code example is worth 1,000 words:

```ruby
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)
```

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.14159265358979</td>
</tr>
<tr>
<td>2.71828182845904</td>
</tr>
<tr>
<td>0.5</td>
</tr>
<tr>
<td>1.0</td>
</tr>
<tr>
<td>2.0</td>
</tr>
<tr>
<td>1.61803398874989</td>
</tr>
</tbody>
</table>

The first thing you noticed was probably the `::` notation. Explaining the *scope operator* (which is what that is) is beyond the...uh...scope of this book. No pun intended. I swear. Suffice it to say, you can use `Math::PI` like it were any other variable.

As you can see, `Math` has all the features you would expect a decent scientific calculator to have. And, as always, the floats are *really close* to being the right answers but not exact; don’t trust them further than you can calculate them.