Mixing It Up

Chapter 4

We've looked at a few different kinds of objects (numbers and letters), and we made variables to point to them; the next thing we want to do is to get them all to play nicely together.

We've seen that if we want a program to print 25, the following does not work, because you can't add numbers and strings:

```
var1 = 2
var2 = '5'

puts var1 + var2
```

Part of the problem is that your computer doesn't know if you were trying to get 7 (2 + 5), or if you wanted to get 25 ('2' + '5').

Before we can add these together, we need some way of getting the string version of var1, or to get the integer version of var2.

Conversions

To get the string version of an object, we simply write .to_s after it:

```
var1 = 2
var2 = '5'

puts var1.to_s + var2
```

```
25
```

Similarly, to_i gives the integer version of an object, and to_f gives the float version. Let's look at what these three methods do (and don't do) a little more closely:

```
var1 = 2
var2 = '5'

puts var1.to_s + var2
puts var1 + var2.to_i
```

```
25
7
```

Notice that, even after we got the string version of var1 by calling to_s, var1 was always pointing at 2, and never at '2'. Unless we explicitly reassign var1 (which requires an = sign), it will point at 2 for the life of the program.

Now let's try some more interesting (and a few just weird) conversions:

```
puts '15'.to_f
puts '99.999'.to_f
puts '99.999'.to_i
puts ''
puts '5 is my favorite number!'.to_i
puts 'Who asked you about 5 or whatever?'.to_i
puts 'Your momma did.'.to_f
puts ''
puts 'stringy'.to_s
puts 3.to_i
```

```
15.0
99.999
99

5
0
0.0

stringy
3
```

So, this probably gave some surprises. The first one is pretty standard, giving 15.0. After that, we converted the string '99.999' to a float and to an integer. The float did what we expected; the integer was, as always, rounded down.

Next, we had some examples of some... unusual strings being converted into numbers. to_i ignores the first thing it doesn't understand, and the rest of the string from that point on. So the first one was converted to 5, but the others, since they started with letters, were ignored completely... so the computer just picks zero.

Finally, we saw that our last two conversions did nothing at all, just as we would expect.

Another Look at puts

There's something strange about our favorite method... Take a look at this:

```
puts 20
puts 20.to_s
puts '20'
```

```
20
```

```
20
20
```

Why do these three all print the same thing? Well, the last two should, since 20.to_s is '20'. But what about the first one, the integer 20? For that matter, what does it even mean to write out the integer 20? When you write a 2 and then a 0 on a piece of paper, you are writing down a string, not an integer. The integer 20 is the number of fingers and toes I have; it isn't a 2 followed by a 0.

Well, here's the big secret behind our friend, puts: Before puts tries to write out an object, it uses to_s to get the string version of that object. In fact, the s in puts stands for string; puts really means put string.

This may not seem too exciting now, but there are many, many kinds of objects in Ruby (you'll even learn how to make your own!), and it's nice to know what will happen if you try to puts a really weird object, like a picture of your grandmother, or a music file or something. But that will come later...

In the meantime, we have a few more methods for you, and they allow us to write all sorts of fun programs...

The Methods gets and chomp

If puts means put string, I'm sure you can guess what gets stands for. And just as puts always spits out strings, gets will only retrieve strings. And whence does it get them?

From you! Well, from your keyboard, anyway. Since your keyboard only makes strings, that works out beautifully. What actually happens is that gets just sits there, reading what you type until you press Enter. Let's try it out:

```
puts gets
```

```
Is there an echo in here?
```

```
Is there an echo in here?
```

Of course, whatever you type in will just get repeated back to you. Run it a few times and try typing in different things.

Now we can make interactive programs! In this one, type in your name and it will greet you:

```
puts 'Hello there, and what\'s your name?'
name = gets
puts 'Your name is ' + name + '?  What a lovely name!'
puts 'Pleased to meet you, ' + name + '.  :)'
```

Eek! I just ran it—I typed in my name, and this is what happened:

```
Hello there, and what's your name?
Chris
Your name is Chris

```

```
?  What a lovely name!
Pleased to meet you, Chris
.  :)
```

Hmmm... it looks like when I typed in the letters C, h, r, i, s, and then pressed Enter, gets got all of the letters in my name and the Enter! Fortunately, there's a method just for this sort of thing: chomp. It takes off any Enters hanging out at the end of your string. Let's try that program again, but with chomp to help us this time:

```
puts 'Hello there, and what\'s your name?'
name = gets.chomp
puts 'Your name is ' + name + '?  What a lovely name!'
puts 'Pleased to meet you, ' + name + '.  :)'
```

```
Hello there, and what's your name?
Chris
Your name is Chris?  What a lovely name!
Pleased to meet you, Chris.  :)
```

Much better! Notice that since name is pointing to gets.chomp, we don't ever have to say name.chomp; name was already chomped.

A Few Things to Try

Write a program which asks for a person's first name, then middle, then last. Finally, it should greet the person using their full name.

Write a program which asks for a person's favorite number. Have your program add one to the number, then suggest the result as a bigger and better favorite number. (Do be tactful about it, though.)

Once you have finished those two programs (and any others you would like to try), let's learn some more (and some more about) methods.