Flow Control

Chapter 6

Ahhhh, flow control. This is where it all comes together. Even though this chapter is shorter and easier than the methods chapter, it will open up a whole world of programming possibilities. After this chapter, we'll be able to write truly interactive programs; in the past we have made programs which say different things depending on your keyboard input, but after this chapter they will actually do different things, too. But before we can do that, we need to be able to compare the objects in our programs. We need...

Comparison Methods

Let's rush through this part so we can get to the next section, Branching, where all the cool stuff happens. So, to see if one object is greater than or less than another, we use the methods > and <, like this:

```
puts 1 > 2
puts 1 < 2
```

```
false
true
```

No problem. Likewise, we can find out if an object is greater-than-or-equal-to another (or less-than-or-equal-to) with the methods >= and <=

```
puts 5 >= 5
puts 5 <= 4
```

```
true
false
```

And finally, we can see if two objects are equal or not using == (which means "are these equal?") and != (which means "are these different?"). It's important not to confuse = with ==. = is for telling a variable to point at an object (assignment), and == is for asking the question: "Are these two objects equal?"

```
puts 1 == 1
puts 2 != 1
```

```
true
true
```

Of course, we can compare strings, too. When strings get compared, they compare their lexicographical ordering, which basically means their dictionary ordering. cat comes before dog in the dictionary, so:

```
puts 'cat' < 'dog'
```

```
true
```

There's a catch, though: the way computers usually do things, they order capital letters as coming before lowercase letters. (That's how they store the letters in fonts, for example: all the capital letters first, then the lowercase ones.) This means that it will think 'Zoo' comes before 'ant', so if you want to figure out which word would come first in a real dictionary, make sure to use downcase (or upcase or capitalize) on both words before you try to compare them.

One last note before Branching: The comparison methods aren't giving us the strings 'true' and 'false'; they are giving us the special objects true and false. (Of course, true.to_s gives us 'true', which is why puts printed 'true'.) true and false are used all the time in...

Branching

Branching is a simple concept, but powerful. In fact, it's so simple that I bet I don't even have to explain it at all; I'll just show you:

```
puts 'Hello, what\'s your name?'
name = gets.chomp
puts 'Hello, ' + name + '.'
if name == 'Chris'
  puts 'What a lovely name!'
end
```

```
Hello, what's your name?
Chris
Hello, Chris.
What a lovely name!
```

But if we put in a different name...

```
Hello, what's your name?
Chewbacca
Hello, Chewbacca.
```

And that is branching. If what comes after the if is true, we run the code between the if and the end. If what comes after the if is false, we don't. Plain and simple.

I indented the code between the if and the end just because I think it's easier to keep track of the branching that way. Almost all programmers do this, regardless of what language they are programming in. It may not seem much help in this simple example, but when things get more complex, it makes a big difference.

Often, we would like a program to do one thing if an expression is true, and another if it is false. That's what else

is for:

```
puts 'I am a fortune-teller.  Tell me your name:'
name = gets.chomp
if name == 'Chris'
  puts 'I see great things in your future.'
else
  puts 'Your future is... Oh my!  Look at the time!'
  puts 'I really have to go, sorry!'
end
```

```
I am a fortune-teller.  Tell me your name:
Chris
I see great things in your future.
```

Now let's try a different name...

```
I am a fortune-teller.  Tell me your name:
Ringo
Your future is... Oh my!  Look at the time!
I really have to go, sorry!
```

Branching is kind of like coming to a fork in the code: Do we take the path for people whose name == 'Chris', or else do we take the other path?

And just like the branches of a tree, you can have branches which themselves have branches:

```
puts 'Hello, and welcome to 7th grade English.'
puts 'My name is Mrs. Gabbard.  And your name is...?'
name = gets.chomp

if name == name.capitalize
  puts 'Please take a seat, ' + name + '.'
else
  puts name + '?  You mean ' + name.capitalize + ', right?'
  puts 'Don\'t you even know how to spell your name?'
  reply = gets.chomp

  if reply.downcase == 'yes'
    puts 'Hmmph!  Well, sit down!'
  else
    puts 'GET OUT!!'
  end
end
```

```
Hello, and welcome to 7th grade English.
My name is Mrs. Gabbard.  And your name is...?
chris
chris?  You mean Chris, right?
Don't you even know how to spell your name?
yes
Hmmph!  Well, sit down!
```

Fine, I'll capitalize it...

```
Hello, and welcome to 7th grade English.
My name is Mrs. Gabbard.  And your name is...?
Chris
Please take a seat, Chris.
```

Sometimes it might get confusing trying to figure out where all of the ifs, elses, and ends go. What I do is write the end at the same time I write the if. So as I was writing the above program, this is how it looked first:

```
puts 'Hello, and welcome to 7th grade English.'
puts 'My name is Mrs. Gabbard.  And your name is...?'
name = gets.chomp

if name == name.capitalize
else
end
```

Then I filled it in with comments, stuff in the code the computer will ignore:

```
puts 'Hello, and welcome to 7th grade English.'
puts 'My name is Mrs. Gabbard.  And your name is...?'
name = gets.chomp

if name == name.capitalize
  # She's civil.
else
  # She gets mad.
end
```

Anything after a # is considered a comment (unless, of course, you are in a string). After that, I replaced the comments with working code. Some people like to leave the comments in; personally, I think well-written code usually speaks for itself. I used to use more comments, but the more "fluent" in Ruby I become, the less I use them. I actually find them distracting much of the time. It's a personal choice; you'll find your own (usually evolving) style. So my next step looked like this:

```
puts 'Hello, and welcome to 7th grade English.'
```

```
puts 'My name is Mrs. Gabbard.  And your name is...?'
name = gets.chomp

if name == name.capitalize
  puts 'Please take a seat, ' + name + '.'
else
  puts name + '?  You mean ' + name.capitalize + ', right?'
  puts 'Don\'t you even know how to spell your name??'
  reply = gets.chomp

  if reply.downcase == 'yes'
  else
  end
end
```

Again, I wrote down the if, else, and end all at the same time. It really helps me keep track of "where I am" in the code. It also makes the job seem easier because I can focus on one small part, like filling in the code between the if and the else. The other benefit of doing it this way is that the computer can understand the program at any stage. Every one of the unfinished versions of the program I showed you would run. They weren't finished, but they were working programs. That way I could test it as I wrote it, which helped to see how it was coming along and where it still needed work. When it passed all of the tests, that's how I knew I was done!

These tips will help you write programs with branching, but they also help with the other main type of flow control:

Looping

Often, you'll want your computer to do the same thing over and over again—after all, that's what computers are supposed to be so good at.

When you tell your computer to keep repeating something, you also need to tell it when to stop. Computers never get bored, so if you don't tell it to stop, it won't. We make sure this doesn't happen by telling the computer to repeat certain parts of a program while a certain condition is true. This works very similarly to how if works:

```
command = ''

while command != 'bye'
  puts command
  command = gets.chomp
end

puts 'Come again soon!'
```

```
Hello?
Hello?
```

```
Hi!
Hi!
Very nice to meet you.
Very nice to meet you.
Oh... how sweet!
Oh... how sweet!
bye
Come again soon!
```

And that's a loop. (You may have noticed the blank line at the beginning of the output; it's from the first puts, before the first gets. How would you change the program to get rid of this first line. Test it! Did it work exactly like the program above, other than that first blank line?)

Loops allow you to do all kinds of interesting things, as I'm sure you can imagine. However, they can also cause problems if you make a mistake. What if your computer gets trapped in an infinite loop? If you think this may have happened, just hold down the Ctrl key and press C.

Before we start playing around with loops, though, let's learn a few things to make our job easier.

A Little Bit of Logic

Let's take a look at our first branching program again. What if my wife came home, saw the program, tried it out, and it didn't tell her what a lovely name she had? Well... she probably wouldn't care. But I'd care! So let's rewrite it:

```
puts 'Hello, what\'s your name?'
name = gets.chomp
puts 'Hello, ' + name + '.'
if name == 'Chris'
  puts 'What a lovely name!'
else
  if name == 'Katy'
    puts 'What a lovely name!'
  end
end
```

```
Hello, what's your name?
Katy
Hello, Katy.
What a lovely name!
```

It works... but it isn't a very pretty program. Why not? Well, the best rule I ever learned in programming was the DRY rule: Don't Repeat Yourself. I could probably write a small book just on why that is such a good rule. In our case, we repeated the line puts 'What a lovely name!'. Why is this such a big deal? Well, what if I made a spelling mistake when I rewrote it? What if I wanted to change it from 'lovely' to 'beautiful' on both lines? I'm lazy, remember? Basically, if I want the program to do the same thing when it gets 'Chris' or 'Katy', then it should

really do the same thing:

```
puts 'Hello, what\'s your name?'
name = gets.chomp
puts 'Hello, ' + name + '.'
if (name == 'Chris' or name == 'Katy')
  puts 'What a lovely name!'
end
```

```
Hello, what's your name?
Katy
Hello, Katy.
What a lovely name!
```

Much better. In order to make it work, I used or. The other logical operators are and and not. It is always a good idea to use parentheses when working with these. Let's see how they work:

```
iAmChris  = true
iAmPurple = false
iLikeFood = true
iEatRocks = false

puts (iAmChris  and iLikeFood)
puts (iLikeFood and iEatRocks)
puts (iAmPurple and iLikeFood)
puts (iAmPurple and iEatRocks)
puts
puts (iAmChris  or iLikeFood)
puts (iLikeFood or iEatRocks)
puts (iAmPurple or iLikeFood)
puts (iAmPurple or iEatRocks)
puts
puts (not iAmPurple)
puts (not iAmChris )
```

```
true
false
false
false

true
true
true
false
```

```
true
false
```

The only one of these which might trick you is or. In English, we often use "or" to mean "one or the other, but not both." For example, your mom might say, "For dessert, you can have pie or cake." She did not mean you could have them both! A computer, on the other hand, uses or to mean "one or the other, or both." (Another way of saying it is, "at least one of these is true.") This is why computers are more fun than moms.

A Few Things to Try

```
"99 bottles of beer on the wall..."
```

Write a program which prints out the lyrics to that beloved classic, that field-trip favorite: "99 Bottles of Beer on the Wall."

Write a Deaf Grandma program. Whatever you say to grandma (whatever you type in), she should respond with HUH?! SPEAK UP, SONNY!, unless you shout it (type in all capitals). If you shout, she can hear you (or at least she thinks so) and yells back, NO, NOT SINCE 1938! To make your program really believable, have grandma shout a different year each time; maybe any year at random between 1930 and 1950. (This part is optional, and would be much easier if you read the section on Ruby's random number generator at the end of the methods chapter.) You can't stop talking to grandma until you shout BYE.

Hint: Don't forget about chomp! 'BYE'with an Enter is not the same as 'BYE' without one!

Hint 2: Try to think about what parts of your program should happen over and over again. All of those should be in your while loop.

Extend your Deaf Grandma program: What if grandma doesn't want you to leave? When you shout BYE, she could pretend not to hear you. Change your previous program so that you have to shout BYE three times in a row. Make sure to test your program: if you shout BYE three times, but not in a row, you should still be talking to grandma.

Leap Years. Write a program which will ask for a starting year and an ending year, and then puts all of the leap years between them (and including them, if they are also leap years). Leap years are years divisible by four (like 1984 and 2004). However, years divisible by 100 are not leap years (such as 1800 and 1900) unless they are divisible by 400 (like 1600 and 2000, which were in fact leap years).

(Yes, it's all pretty confusing, but not as confusing as having July in the middle of the winter, which is what would eventually happen.)

When you finish those, take a break! You've learned a lot already. Congratulations! Are you surprised at the number of things you can tell a computer to do? A few more chapters and you'll be able to program just about anything. Seriously! Just look at all the things you can do now that you couldn't do without looping and branching.

Now let's learn about a new kind of object, one which keeps track of lists of other objects: arrays.