We’ve covered a lot of the basics, but this is where we really breathe life into our programs. Up to this point, our programs have been so flat and predictable. Each time we run them, we’ll get pretty much the same experience. I mean, if a program asks me for my name, I guess instead of “Chris,” I could say “Stud-chunks McStallion” (as they used to call me), but that’s hardly a new experience.

After this chapter, though, we’ll be able to write truly interactive programs. In the past, we made programs that said different things depending on your keyboard input, but after this chapter they will actually do different things. But how will we determine when to do one thing instead of another? We need...

7.1 Comparison Methods

You’re getting good at this, so I’ll try to let the code do the talking. First, to see whether one object is greater than or less than another, we use the methods > and <:

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

```
false
true
```

No problem.

Likewise, we can find out whether an object is greater than or equal to another (or less than or equal to) with the methods >= and <=:

```ruby
puts 5 >= 5
puts 5 <= 4
```
And finally, we can see whether two objects are equal 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?”

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

```
true
false
```

Of course, we can compare strings, too. When strings get compared, Ruby compares their lexicographical ordering, which basically means the order they appear in a dictionary. For example, `cat` comes before `dog` in the dictionary, so we have this:

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

```
true
```

This has 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 and then the lowercase ones.) This means it will think `'Xander'` comes before `'bug lady'`. 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.

```ruby
puts 'bug lady'.downcase < 'Xander'.downcase
```

```
false
true
```

Similarly surprising is this:

```ruby
puts 2 < 10
puts '2' < '10'
```

```
true
false
```

OK, 2 is less than 10, so no problem. But that last one? Well, the `'1'` character comes before the `'2'` character—remember, in a string those are just characters. The `'0'` character after the `'1'` doesn’t make the `'1'` any larger.
One last note before we move on: the comparison methods aren’t giving us the strings ‘true’ and ‘false’; they are giving us the special objects true and false that represent...well, truth and falsity. (Of course, true.to_s gives us the string ‘true’, which is why puts printed true.) true and false are used all the time in a language construct called branching.

7.2 Branching

Branching is a simple concept, but it’s 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:

```ruby
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 that helpful in this simple example, but when programs get more complex, it makes a big difference. Often, when people send me programs that don’t work but they can’t figure out why, it’s something that is both:

- obvious to see what the problem is if the indentation is nice, and
- impossible to see what the problem is otherwise.

So, try to keep your indentation nice and consistent. Have your if and end line up vertically, and have everything between them indented. I use an indentation of two spaces.
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

伊拉姆 a fortune-teller. Tell me your name:
➾ Chris
伊拉姆 see great things in your future.

Now let’s try a different name:

伊拉姆 a fortune-teller. Tell me your name:
➾ Boromir
伊拉姆 your future is...oh my! Look at the time!
伊拉姆 really have to go, sorry!

And one more:

伊拉姆 a fortune-teller. Tell me your name:
➾ Ringo
伊拉姆 your future is...oh my! Look at the time!
伊拉姆 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, less fortuitous, path? (Well, I guess you could also call it the path of fame, fortune, and glory. But it’s my fortune-teller, and I say it’s less fortuitous. So there.) Clearly, branching can get pretty deep.

Just like the branches of a tree, you can have branches that themselves have branches, as we can see on the next page.

```
puts 'Hello, and welcome to seventh 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!'`
```ruby
  else
    puts 'GET OUT!!'
  end
end

>Hello, and welcome to seventh 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 my name:

>Hello, and welcome to seventh 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 the if's, else's, and end's go. What I do is write the end at the same time I write the if. So, as I was writing the previous program, this is how it looked first:

```ruby
puts 'Hello, and welcome to seventh grade English.'
puts 'My name is Mrs. Gabbard. And your name is....?'
nname = gets.chomp
if nname == nname.capitalize
else
end
```

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

```ruby
puts 'Hello, and welcome to seventh grade English.'
puts 'My name is Mrs. Gabbard. And your name is....?'
nname = gets.chomp
if nname == nname.capitalize
  # She's civil.
else
  # She gets mad.
end
```

Anything after a # is considered a comment (unless, of course, the # is 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. (The trick, of course, is in writing well-written code.) 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.
Anyway, my next step looked like this:

```ruby
puts 'Hello, and welcome to seventh 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 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, such as 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 them as I wrote them, which helped me see how my program was coming along and where it still needed work. When it passed all the tests, I knew I was done.

I strongly suggest you approach your programs in this way. These tips will help you write programs with branching, but they also help with the other main type of flow control.

### 7.3 Looping

Often, you’ll want your computer to do the same thing over and over again. After all, that’s what they’re supposed to be good at doing.

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 when 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. It works the way if works:

```ruby
input = ''
while input != 'bye'
  puts input
  input = gets.chomp
end
puts 'Come again soon!
```
It's not a fabulous program, though. For one thing, while tests your condition at the top of the loop. In this case we had to tweak our loop so it could test there. This made us puts a blank line before we did our first gets. In my mind, it just feels like the gets comes first and the echoing puts comes later. It'd be nice if we could say something like this:

```ruby
# THIS IS NOT A REAL PROGRAM!
while just_like_go_forever
  input = gets.chomp
  puts input
  if input == 'bye'
    stop_looping
  end
end
puts 'Come again soon!'
```

That's not valid Ruby code, but it's close! To get it to loop forever, we just need to give while a condition that's always true. And Ruby does have a way to break out of a loop:

```ruby
# THIS IS SO TOTALLY A REAL PROGRAM!
while 'Spike' > 'Angel'
  input = gets.chomp
  puts input
  if input == 'bye'
    break
  end
end
puts 'Come again soon!'
```

It's a little more complicated, but it gets the job done.
Now, isn't that better? OK, I'll admit, the 'Spike' > 'Angel' thing is a little silly. When I get bored coming up with jokes for these examples, I'll usually just use the actual true object:

```ruby
while true
  input = gets.chomp
  puts input
  if input == 'bye'
    break
  end
end
puts 'Come again soon!' 
```

Now that's a loop. It's considerably trickier than a branch, so take a minute to look it over and let it sink in....

Loops are lovely things. However, like high-maintenance girlfriends or bubble gum, they can cause big problems if handled improperly. Here's a big one: what if your computer gets trapped in an infinite loop? If you think this may have happened, just go to your command line, hold down the **Ctrl** key, and press **C**. (You are running these from the command line, right?)

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

## 7.4 A Little Bit of Logic

Let's take another look at our first branching program, on page 39. What if my wife came home, saw the program, tried it, and it didn't tell her what a lovely name she had? I wouldn't want her to flip out, so let's rewrite it:

```ruby
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!

Well, it works...but it isn’t a very pretty program. Why not? It just doesn’t feel right to me that the whole “Katy” chunk of code is not lined up with the “Chris” chunk of code. These are supposed to be totally equal and symmetrical options, yet one feels distinctly subordinate to the other. (In fact, this code would probably get me sleeping on the couch faster than just leaving her out of the program altogether.) This code just isn’t jiving with my mental model.

Fortunately, another Ruby construct can help: `elsif`. This code means the same thing as the last program but feels so much lovelier:

```ruby
puts 'Hello, what\'s your name?
name = gets.chomp
puts 'Hello, ' + name + '.
if   name == 'Chris'
    puts 'What a lovely name!'  # Here we add a space and newline
elsif name == 'Katy'
    puts 'What a lovely name!'  # Here we add a space and newline
end

Hello, what's your name?

➾ Katy

Hello, Katy.
What a lovely name!
```

This is a definite improvement, but something is still wrong. If I want the program to do the same thing when it gets `Chris` or `Katy`, then it should really do the same thing, as in execute the same code. Here we have two different lines of code doing the same thing. That’s not right. That’s not how I’m thinking about this.

More pragmatically, it’s just a bad idea to duplicate code anywhere. Remember the DRY rule? Don’t Repeat Yourself! For pragmatic reasons, for aesthetic reasons, or just because you’re lazy, don’t ever repeat yourself! Weed out duplication in code (or even design) whenever you see it. In our case, we repeated the line `puts 'What a lovely name!'`. What we’re trying to say is just, “If the name is `Chris` or `Katy`, do this.” Let’s just code it that way:
puts 'Hello, what\'s your name?'
name = gets.chomp
puts 'Hello, ' + name + '.'
if name == 'Chris' || name == 'Katy'
  puts 'What a lovely name!'  
end

>Hello, what\'s your name?

Katy

>Hello, Katy.
What a lovely name!

Nice. Much, much better. And it\'s even shorter! I don\'t know about you, but I\'m excited. It\'s almost the same as the original program! Bliss, I tell you...sparkly programming bliss!

To make it work, I used ||, which is how we say “or” in most programming languages.

At this point, you might be wondering why we couldn\'t just say this:

...  
if name == ('Chris' || 'Katy')
  puts 'What a lovely name!'  
end

It makes sense in English, but you have to remember how staggeringly brilliant humans are compared to computers. The reason this makes sense in English is that humans are just fabulous at dealing with context. In this context, it\'s clear to a human that “if your name is Chris or Katy” means “if your name is Chris or if it is Katy.” (I even used “it”—another triumph of human context handling.) But when your computer sees ('Chris' || 'Katy'), it\'s not even looking at the name == code; before it gets there, it just tries to figure out whether one of 'Chris' or 'Katy' is true...because that\’s what || does. But that doesn\'t really make sense, so you have to be explicit and write the whole thing.

Anyway, that\’s “or.” The other logical operators are && (“and”) and ! (“not”). Let\’s see how they work:

i_am_chris  = true
i_am_purple = false
i_like_beer = true
i_eat_rocks = false

puts i_am_chris  && i_like_beer
puts i_like_beer && i_eat_rocks
puts i_am_purple && i_like_beer
puts i_am_purple && i_eat_rocks
puts
puts i_am_chris || i_like_beer
puts i_like_beer || i_eat_rocks
puts i_am_purple || i_like_beer
puts i_am_purple || i_eat_rocks
puts !i_am_purple
puts !i_am_chris
true
false
false
false
true
true
true
false
true
false
true
false
true
false
true
false
true
false
The only one of these that might trick you is ||. 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 || 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. (Obviously I think my mom is far less likely to read this book than my wife is.)

Just to make sure everything is well cemented for you, let’s look at one more example before you go it alone. This will be a simulation of talking to my son, C, back when he was 2. (Just for background, when he talks about Ruby, Nono, and Emma, he is referring to his baby sister, Ruby, and his friends Giuliano and Emma. He manages to bring everyone he loves into every conversation he has. And yes, we did name our children after programming languages. And yes, my wife is the coolest woman ever.) So, without further ado, this is pretty much what happens whenever you ask C to do something:

```ruby
while true
  puts 'What would you like to ask C to do?'
  request = gets.chomp
  puts 'You say, "C, please ' + request + '"'
  puts 'C\'s response:'
  puts '"C ' + request + '."
  puts '"Papa ' + request + ', too."
  puts '"Mama ' + request + ', too."
  puts '"Ruby ' + request + ', too."'
```
puts '"Nono ' + request + ', too."'
puts '"Emma ' + request + ', too."'
puts
if request == 'stop'
  break
end
end

Let's chat with C a bit on the next page.

What would you like to ask C to do?

→ eat

You say, "C, please eat"
C’s response:
"C eat."
"Papa eat, too."
"Mama eat, too."
"Ruby eat, too."
"Nono eat, too."
"Emma eat, too."

What would you like to ask C to do?

→ go potty

You say, "C, please go potty"
C’s response:
"C go potty."
"Papa go potty, too."
"Mama go potty, too."
"Ruby go potty, too."
"Nono go potty, too."
"Emma go potty, too."

What would you like to ask C to do?

→ hush

You say, "C, please hush"
C’s response:
"C hush."
"Papa hush, too."
"Mama hush, too."
"Ruby hush, too."
"Nono hush, too."
"Emma hush, too."

What would you like to ask C to do?

→ stop

You say, "C, please stop"
C’s response:
"C stop."
"Papa stop, too."
"Mama stop, too."
"Ruby stop, too."
"Nono stop, too."
"Emma stop, too."

Yeah, that’s about what it was like. You couldn’t sneeze without hearing about Emma or Nono sneezing, too.

7.5 A Few Things to Try

• “99 Bottles of Beer on the Wall.” Write a program that prints out the lyrics to that beloved classic, “99 Bottles of Beer on the Wall.”

• Deaf grandma. Whatever you say to Grandma (whatever you type in), she should respond with this:

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 on page 33.) You can’t stop talking to Grandma until you shout BYE.

Hint 1: Don’t forget about chomp! ‘BYE’ with an Enter at the end 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.

Hint 3: People often ask me, “How can I make rand give me a number in a range not starting at zero?” Well, you can’t; rand just doesn’t work that way. So, I guess you’ll have to do something to the number rand returns to you.

• Deaf grandma extended. 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 that asks for a starting year and an ending year and then puts all the leap years between them (and including them,
if they are also leap years). Leap years are years divisible by 4 (like 1984 and 2004). However, years divisible by 100 are not leap years (such as 1800 and 1900) unless they are also divisible by 400 (such as 1600 and 2000, which were in fact leap years). What a mess!

When you finish those, take a break! That was a lot of programming. Congratulations! You’re well on your way. Relax, have a nice cold (possibly root) beer, and continue tomorrow.