Exercise 11: Asking Questions

Now it is time to pick up the pace. I have got you doing a lot of printing so that you get used to typing simple things, but those simple things are fairly boring. What we want to do now is get data into your programs. This is a little tricky because you have learn to do two things that may not make sense right away, but trust me and do it anyway. It will make sense in a few exercises.

Most of what software does is the following:

1. Take some kind of input from a person.
2. Change it.
3. Print out something to show how it changed.

So far you have only been printing, but you haven't been able to get any input from a person, or change it. You may not even know what "input" means, so rather than talk about it, let's have you do some and see if you get it. Next exercise we'll do more to explain it.

```ruby
print "How old are you? "
age = gets.chomp()
print "How tall are you? "
height = gets.chomp()
print "How much do you weigh? "
weight = gets.chomp()

puts "So, you're #{age} old, #{height} tall and #{weight} heavy."
```

Note: Notice that we are using `print` instead of `puts` to do the prompting. `print` doesn't add a new line automatically, so your answer can go on the same line as the question. `puts` on the other hand, adds a newline automatically.

What You Should See

```
$ ruby ex11.rb
How old are you? 35
How tall are you? 6'2"
How much do you weigh?  180lbs
So, you're 35 old, 6'2" tall and 180lbs heavy.
$
```

Extra Credit

1. Go online and find out what Rubys `gets` and `chomp` methods do.
2. Can you find other ways to use `gets.chomp`? Try some of the samples you find.
3. Write another "form" like this to ask some other questions.

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

```
false
true
```

Similarly surprising is this:

```
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*, and that's a big enough topic that we need a fresh new page just to hold it.

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:

```
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.

```ruby
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:
Boromir
Your future is...oh my!  Look at the time!
I really have to go, sorry!
```

And one more:

```
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, 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!'
  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:

```
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
else
end
```

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

```
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
  #  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:

```
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. This works kind of like how **if** works:

```
input = ''

while input != 'bye'
  puts input
  input = 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!
```

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:

```
#  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:

```
#  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!'
```

```
Hi, and your name is...
Hi, and your name is...
Cute.  And original.
Cute.  And original.
What, are you like... my little brother?!
What, are you like... my little brother?!
bye
bye
Come again soon!
```

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:

```
while true
  input = gets.chomp
  puts input
  if input == 'bye'
    break
  end
end

puts 'Come again soon!'
```

```
Hey.
Hey.
You again?!
You again?!
bye
bye
Come again soon!
```

And 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 43. 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:

Mind you, Katy is as lovely and sweet as she is likely to read this book, so I feel I should point out that she would never flip out about something like this. She saves that for when I've done something really horrible, like lose one of her James Bond DVDs.

```
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!'
elsif name == 'Katy'
  puts 'What a lovely name!'
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
puts !i_am_purple
puts !i_am_chris
```

```
true
false
false
false

true
true
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 38.) 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.

Exercise 31: Making Decisions

In the first half of this book you mostly just printed out things and called functions, but everything was basically in a straight line. Your scripts ran starting at the top, and went to the bottom where they ended. If you made a function you could run that function later, but it still didn't have the kind of branching you need to really make decisions. Now that you have `if`, `else`, and `elsif` you can start to make scripts that decide things.

In the last script you wrote out a simple set of tests asking some questions. In this script you will ask the user questions and make decisions based on their answers. Write this script, and then play with it quite a lot to figure it out.

```ruby
def prompt
  print "> "
end

puts "You enter a dark room with two doors.  Do you go through door #1 or door #2?"

prompt; door = gets.chomp

if door == "1"
  puts "There's a giant bear here eating a cheese cake.  What do you do?"
  puts "1. Take the cake."
  puts "2. Scream at the bear."

  prompt; bear = gets.chomp

  if bear == "1"
    puts "The bear eats your face off.  Good job!"
  elsif bear == "2"
    puts "The bear eats your legs off.  Good job!"
  else
    puts "Well, doing #{bear} is probably better.  Bear runs away."
  end

elsif door == "2"
  puts "You stare into the endless abyss at Cthuhlu's retina."
  puts "1. Blueberries."
  puts "2. Yellow jacket clothespins."
  puts "3. Understanding revolvers yelling melodies."

  prompt; insanity = gets.chomp

  if insanity == "1" or insanity == "2"
    puts "Your body survives powered by a mind of jello.  Good job!"
  else
    puts "The insanity rots your eyes into a pool of muck.  Good job!"
  end

else
```

```ruby
  puts "You stumble around and fall on a knife and die.  Good job!"
end
```

A key point here is that you are now putting the `if-statements` *inside* `if-statements` as code that can run. This is very powerful and can be used to create "nested" decisions, where one branch leads to another and another.

Make sure you understand this concept of `if-statements` inside `if-statements`. In fact, do the extra credit to really nail it.

What You Should See

Here is me playing this little adventure game. I do not do so well.

```
$ ruby ex31.rb
You enter a dark room with two doors.  Do you go through door #1 or door #2?
> 1
There's a giant bear here eating a cheese cake.  What do you do?
1. Take the cake.
2. Scream at the bear.
> 2
The bear eats your legs off.  Good job!

$ ruby ex31.rb
You enter a dark room with two doors.  Do you go through door #1 or door #2?
> 1
There's a giant bear here eating a cheese cake.  What do you do?
1. Take the cake.
2. Scream at the bear.
> 1
The bear eats your face off.  Good job!

$ ruby ex31.rb
You enter a dark room with two doors.  Do you go through door #1 or door #2?
> 2
You stare into the endless abyss at Cthuhlu's retina.
1. Blueberries.
2. Yellow jacket clothespins.
3. Understanding revolvers yelling melodies.
> 1
Your body survives powered by a mind of jello.  Good job!

$ ruby ex31.rb
You enter a dark room with two doors.  Do you go through door #1 or door #2?
> 2
You stare into the endless abyss at Cthuhlu's retina.
1. Blueberries.
2. Yellow jacket clothespins.
3. Understanding revolvers yelling melodies.
> 3
The insanity rots your eyes into a pool of muck.  Good job!

$ ruby ex31.rb
You enter a dark room with two doors.  Do you go through door #1 or door #2?
> stuff
You stumble around and fall on a knife and die.  Good job!

$ ruby ex31.rb
```

```
You enter a dark room with two doors.  Do you go through door #1 or door #2?
> 1
There's a giant bear here eating a cheese cake.  What do you do?
1. Take the cake.
2. Scream at the bear.
> apples
Well, doing apples is probably better.  Bear runs away.
```

Extra Credit

Make new parts of the game and change what decisions people can make. Expand the game out as much as you can before it gets ridiculous.

Chapter 10

There's Nothing New to Learn in Chapter 10

Congratulations! You're a programmer! At this point we've covered most of the basics of programming. The rest of the book is pretty much just fleshing things out for you, showing a few tricks, presenting ways to save time, and so on.

But it hasn't been easy, I imagine. If your brain isn't already hurting by this point, either you're brilliant, you were already a programmer before picking up this book, or you do not yet comprehend the power (and terror) of what you've just learned.

Since you've done so well making it this far, I'll make you a deal: we won't cover anything new in this chapter! We'll just sort of...reminisce.

This should make all of our lives a bit easier. :)

10.1 Recursion

You know how to make methods, and you know how to call methods. (Your very first program did that! Ahhh, those simple days of one-line programs....) When you write methods, you'll usually fill them with method calls. You can make methods, and they can call methods...see where I'm going with this? Yeah? No?

What if you wrote a method that called itself?

That's *recursion.*

Well, on the surface, it's an absurd idea; if all a method did was call itself, which would just call itself again, it would loop like that forever.

(Although this is not technically a loop, it is similar; we can usually replace loops with recursion if we feel like it.) But of course, it could do other things as well and maybe call itself only some of the time.

Let's look at what our ask method from our psych program would look like with recursion instead of **while** loops:

```
def ask_recursively question
  puts question
  reply = gets.chomp.downcase

  if    reply == 'yes'
    true
  elsif reply == 'no'
    false
  else
    puts 'Please answer "yes" or "no".'
    ask_recursively question  #  This is the magic line.
  end
end

ask_recursively 'Do you wet the bed?'
```

```
Do you wet the bed?
no way!
Please answer "yes" or "no".
Do you wet the bed?
NO, dude!
Please answer "yes" or "no".
Do you wet the bed?
I said, "NO!"
Please answer "yes" or "no".
Do you wet the bed?
NOOOOOOOOOOOOOOOOOOOO!!!!!
Please answer "yes" or "no".
Do you wet the bed?
nonononononononono
Please answer "yes" or "no".
Do you wet the bed?
<gasp>
```

```
Please answer "yes" or "no".
Do you wet the bed?
yes
```

Oh, *nice!* That is smooth, with a capital *Smooth*...er, as they say. Wow. Now I feel kind of bad about pushing that sorry loop version onto you in the previous chapter. This one is super short, has no unnecessary variables, and has no returns; it just does what it does.

As who said? Ten points if you know.

Honestly, I'm a little surprised at how nice that was. I would not normally have thought of using recursion here. In general, I try to use loops when I'm going to be doing the same thing over and over again, and I use recursion when a small part of the problem resembles the whole problem; the classic example is in computing factorials. Maybe I should think about using recursion more often....

Anyway, since I brought them up and since there seems to be some universal law that every introduction to recursion involves computing factorials, we might as well give it a whirl. I'm feeling pretty rebellious, anyway, for not using factorials as my *first* recursion example, so look at this before the recursion police take me away:

```ruby
def factorial num
  if num < 0
    return 'You can\'t take the factorial of a negative number!'
  end

  if num <= 1
    1
  else
    num * factorial(num-1)
  end
end

puts factorial(3)
puts factorial(30)
```

```
6
265252859812191058636308480000000
```

There you are: factorials. For those of you who had better things to do than go to math class (clearly I did not), the factorial of an integer is the product of all the integers from itself down to 1. In other words, the factorial of 3 (written 3!, as if to fool you into thinking factorials are really exciting) is just 3 times 2 times 1, or 6. And 0! is 1 (I could give you a "sound of one hand clapping" sort of argument you may or may not find satisfying, or you could just take my word for it), and the factorial of a negative number is just plain bad sportsmanship.

But these examples have been sort of contrived (though I did end up really liking how ask_recursively turned out). How about a real example?

When I was generating the worlds for the game Civilization III, I wanted worlds with two primary supercontinents; those tend to be a lot of fun and just sort of feel "earthy" and...*real*. So after I generated the land masses (which was some pretty clever programming there, too), I wanted to test them to see what the sizes of the different continents were. If there were two of relatively equal size (say, differing by a factor of 2 or less) and no others close in size, I'd say that was a pretty good world.

The process, then, was something like the following:

1. Build the world.
2. Find a "continent" (which could be a one-tile island...at this point I wouldn't know).
3. Compute its size.
4. Find another continent (making sure not to count any of them twice but also making sure each gets counted), and repeat the process.
5. Then find the largest two, and see whether they look like fun to play on.

The fun part (actually, it was all fun, not just this part!) was in computing each continent's size, because the best way to do that was recursively.

Let's look at a trimmed-down version. Let's say we have an 11x11 world (represented as an array of arrays...basically just a grid) and that we want to find the size of the continent in the middle (that is, the continent of which tile (5,5) is a part). We don't want to count any land tiles belonging to any of the other continents. Also, as in Civilization III, we'll say that tiles touching only at the corners are still considered to be on the same continent (since units could walk along diagonals).

But before we get to the code, let's solve the problem in English first. My initial plan was to look at every tile on the map, and if that tile is a land tile on the continent I'm looking for, we add 1 to the running total. The problem, though, is how do we know whether a land tile is on the same continent as some other land tile? There are ways to solve this problem, but they all seemed to be too messy; either I was keeping track of lots of information I didn't feel like I needed or I seemed to be doing the same computation over and over again.

But then I thought, hey, two tiles are on the same continent if you can walk from one to the other. (That was essentially the operating definition of continent in Civilization III.) So that's how the code should work! First, you count the spot you are standing on (duh); in this case, that means tile (5,5). Then, you send out eight little guys, one in each direction, and tell them to count the rest of the continent in that direction. The only rule is that no one can count a tile that someone else has already counted. When those eight guys return, you add their answers to your already-running total (which is just 1, from the tile you started with), and that's your answer.

Brilliant! Except for one tiny little detail...how are those eight little guys supposed to determine the size of the continent? You just shrugged the problem onto them! The only tile you counted was the one you were standing on. This is pretty frickin' lazy. Which is probably a good thing....

How are your eight little helpers supposed to compute the size of the continent? The same way you do! So somehow, by a bunch of little, lazy, imaginary helpers counting only the tile they are on, you get the size of the whole continent. (We still need to make sure no tile is counted twice, but we can just mark each tile as it is visited to keep track.) Without further ado, behold the magic of recursion.

(OK, so there *is* some ado. But just a little. Only a soupçon of ado. And that's only because I want to display all the code on a single page spread, and there isn't enough room on this page, so I have to fill up the space with this paragraph so the code will start at the top of the next page.)

```
#  These are just to make the map
#  easier for me to read.  "M" is
#  visually more dense than "o".
M = 'land'
o = 'water'

world = [[o,o,o,o,o,o,o,o,o,o,o],
         [o,o,o,o,M,M,o,o,o,o,o],
         [o,o,o,o,o,o,o,o,M,M,o],
         [o,o,o,M,o,o,o,o,o,M,o],
         [o,o,o,M,o,M,M,o,o,o,o],
         [o,o,o,o,M,M,M,M,o,o,o],
         [o,o,o,M,M,M,M,M,M,M,o],
         [o,o,o,M,M,o,M,M,M,o,o],
         [o,o,o,o,o,o,M,M,o,o,o],
         [o,M,o,o,o,M,o,o,o,o,o],
         [o,o,o,o,o,o,o,o,o,o,o]]

def continent_size world, x, y
  if world[y][x] != 'land'
    #  Either it's water or we already
    #  counted it, but either way, we don't
    #  want to count it now.
    return 0
  end

  #  So first we count this tile...
  size = 1
  world[y][x] = 'counted land'

  #  ...then we count all of the
  #  neighboring eight tiles (and,
  #  of course, their neighbors by
  #  way of the recursion).
  size = size + continent_size(world, x-1, y-1)
  size = size + continent_size(world, x  , y-1)
  size = size + continent_size(world, x+1, y-1)
  size = size + continent_size(world, x-1, y  )
  size = size + continent_size(world, x+1, y  )
  size = size + continent_size(world, x-1, y+1)
  size = size + continent_size(world, x  , y+1)
  size = size + continent_size(world, x+1, y+1)

  size
end

puts continent_size(world, 5, 5)
```

Drumroll, please....

```
23
```

And there you have it. Even if the world was much, much larger and the continent was totally bizarre and oddly shaped, it would still work just fine. Well, there is actually one small bug for you to fix. This code works fine because the continent does not border the edge of the world. If it did, then when we send our little guys out (that is, call continent_size on a new tile), some of them would fall off the edge of the world (that is, call continent_size with invalid values for x and/or y), which would probably crash on the very first line of the method.

It seems like the obvious way to fix this would be to do a check before each call to continent_size (sort of like sending your little guys out only if they aren't going to fall over the edge of the world), but that would require eight separate (yet nearly identical) checks in your method. Yuck. It would be lazier to just send your guys out and have them shout back "ZERO!" if they fall off the edge of the world. (In other words, put the check right at the top of the method, very much like the check we put in to see whether the tile was uncounted land.) Go for it! Of course, you'll have to make sure it works; test it by extending the continent to touch one (or better yet, all four) of the edges of the world.

And that, my friends, is recursion. It's not really anything new, just a new way of thinking of the same old stuff you already learned.

10.2 Rite of Passage: Sorting

Remember the sorting program you wrote on page 65 where you asked for a list of words, sorted it, and then displayed the sorted list? The program was made much easier because you used the array's sort method. But, like the Jedi who constructs his own lightsaber, you'll exhibit a greater mastery if you write your own sorting method. Hey, we've all done it. It's not easy, but this kind of problem solving is part of nearly every program you'll write, so you'd best get your practice now.

But where do you begin? Much like with continent_size, it's probably best to try to solve the problem in English first. Then translate it into Ruby when you've wrapped your head around it.

So, we want to sort an array of words, and we know how to find out which of two words comes first in the dictionary (using <).

What strikes me as probably the easiest way to do this is to keep two more lists around: one will be our list of already-sorted words, and the other will be our list of still-unsorted words. We'll take our list of words, find the "smallest" word (that is, the word that would come first in the dictionary), and stick it at the end of the already-sorted list. All of the other words go into the still-unsorted list. Then you do the same thing again but using the still-unsorted list instead of your original list: find the smallest word, move it to the sorted list, and move the rest to the unsorted list. Keep going until your still-unsorted list is empty.

That doesn't sound too bad, but it's keeping all of the details straight that makes it so tricky. Go ahead and try it, and see how it looks. In fact, try it twice: once using recursion and once without. With the recursion, you might need a *wrapper method*, a tiny method that wraps up another method into a cute little package, like this:

```
def sort some_array  #  This "wraps" recursive_sort.
  recursive_sort some_array, []
end

def recursive_sort unsorted_array, sorted_array
  #  Your fabulous code goes here.
end
```

What was the point of the wrapper method? Well, recursive_sort took two parameters, but if you were just trying to sort an array, you would always have to pass in an empty array as the second parameter. This is a silly thing to have to remember. Here, the wrapper method passes it in for us, so we never have to think about it again.

When you're done, make sure to test your code! Type in duplicate words and things like that. A great way to test would be to use the built-in sort method to get a sorted version of your list right away. Then, after you have sorted it for yourself, make sure the two lists are equal.

10.3 A Few Things to Try

- *Shuffle.* Now that you've finished your new sorting algorithm, how about the opposite? Write a shuffle method that takes an array and returns a totally shuffled version. As always, you'll want to test it, but testing this one is trickier: How can you test to make sure you are getting a perfect shuffle? What would you even say a perfect shuffle would be? Now test for it.
- *Dictionary sort.* Your sorting algorithm is pretty good, sure. But there was always that sort of embarrassing point you were hoping I'd just sort of gloss over, right? About the capital letters? Your sorting algorithm is good for general-purpose sorting, but when you sort strings, you are using the ordering of the characters in your fonts (called the *ASCII codes*) rather than true dictionary ordering. In a dictionary, case (upper or lower) is irrelevant to the ordering. So, make a new method to sort words (something like dictionary_sort). Remember, though, that if I give your program words starting with capital letters, it should return words with those same capital letters, just ordered as you'd find in a dictionary.

10.4 One More Example

I think another example method would be helpful here. We'll call this one english_number. It will take a number, like 22, and return the English version of it (in this case, the string 'twenty-two'). For now, let's have it work only on integers from 0 to 100:

```
def english_number number
  #  We accept numbers from 0 to 100.
  if number < 0
    return 'Please enter a number zero or greater.'
  end
  if number > 100
    return 'Please enter a number 100 or less.'
  end
```

```
num_string = ''  #  This is the string we will return.

#  "left"  is how much of the number
#          we still have left to write out.
#  "write" is the part we are
#          writing out right now.
#  write and left... get it?  :)
left  = number
write = left/100          #  How many hundreds left?
left  = left - write*100  #  Subtract off those hundreds.

if write > 0
  return 'one hundred'
end

write = left/10          #  How many tens left?
left  = left - write*10  #  Subtract off those tens.

if write > 0
  if write == 1  #  Uh-oh...
    #  Since we can't write "tenty-two"
    #  instead of "twelve", we have to
    #  make a special exception for these.
    if    left == 0
      num_string = num_string + 'ten'
    elsif left == 1
      num_string = num_string + 'eleven'
    elsif left == 2
      num_string = num_string + 'twelve'
    elsif left == 3
      num_string = num_string + 'thirteen'
    elsif left == 4
      num_string = num_string + 'fourteen'
    elsif left == 5
      num_string = num_string + 'fifteen'
    elsif left == 6
      num_string = num_string + 'sixteen'
    elsif left == 7
      num_string = num_string + 'seventeen'
    elsif left == 8
      num_string = num_string + 'eighteen'
```