Writing Your Own Methods

Chapter 8

As we've seen, loops and iterators allow us to do the same thing (run the same code) over and over again. However, sometimes we want to do the same thing a number of times, but from different places in the program. For example, let's say we were writing a questionnaire program for a psychology student. From the psychology students I have known and the questionnaires they have given me, it would probably go something like this:

```ruby
puts 'Hello, and thank you for taking the time to'
puts 'help me with this experiment.  My experiment'
puts 'has to do with the way people feel about'
puts 'Mexican food.  Just think about Mexican food'
puts 'and try to answer every question honestly,'
puts 'with either a "yes" or a "no".  My experiment'
puts 'has nothing to do with bed-wetting.'
puts

### We ask these questions, but we ignore their answers.

goodAnswer = false
while (not goodAnswer)
  puts 'Do you like eating tacos?'
  answer = gets.chomp.downcase
  if (answer == 'yes' or answer == 'no')
    goodAnswer = true
  else
    puts 'Please answer "yes" or "no".'
  end
end

goodAnswer = false
while (not goodAnswer)
  puts 'Do you like eating burritos?'
  answer = gets.chomp.downcase
  if (answer == 'yes' or answer == 'no')
    goodAnswer = true
  else
    puts 'Please answer "yes" or "no".'
  end
end

### We pay attention to *this* answer, though.
```

```
goodAnswer = false
while (not goodAnswer)
  puts 'Do you wet the bed?'
  answer = gets.chomp.downcase
  if (answer == 'yes' or answer == 'no')
    goodAnswer = true
    if answer == 'yes'
      wetsBed = true
    else
      wetsBed = false
    end
  else
    puts 'Please answer "yes" or "no".'
  end
end

goodAnswer = false
while (not goodAnswer)
  puts 'Do you like eating chimichangas?'
  answer = gets.chomp.downcase
  if (answer == 'yes' or answer == 'no')
    goodAnswer = true
  else
    puts 'Please answer "yes" or "no".'
  end
end

puts 'Just a few more questions...'

goodAnswer = false
while (not goodAnswer)
  puts 'Do you like eating sopapillas?'
  answer = gets.chomp.downcase
  if (answer == 'yes' or answer == 'no')
    goodAnswer = true
  else
    puts 'Please answer "yes" or "no".'
  end
end

### Ask lots of other questions about Mexican food.

puts
puts 'DEBRIEFING:'
puts 'Thank you for taking the time to help with'
puts 'this experiment.  In fact, this experiment'
puts 'has nothing to do with Mexican food.  It is'
puts 'an experiment about bed-wetting.  The Mexican'
puts 'food was just there to catch you off guard'
puts 'in the hopes that you would answer more'
puts 'honestly.  Thanks again.'
puts
puts wetsBed
```

```
Hello, and thank you for taking the time to
help me with this experiment.  My experiment
has to do with the way people feel about
Mexican food.  Just think about Mexican food
and try to answer every question honestly,
with either a "yes" or a "no".  My experiment
has nothing to do with bed-wetting.

Do you like eating tacos?
yes
Do you like eating burritos?
yes
Do you wet the bed?
no way!
Please answer "yes" or "no".
Do you wet the bed?
NO
Do you like eating chimichangas?
yes
Just a few more questions...
Do you like eating sopapillas?
yes

DEBRIEFING:
Thank you for taking the time to help with
this experiment.  In fact, this experiment
has nothing to do with Mexican food.  It is
an experiment about bed-wetting.  The Mexican
food was just there to catch you off guard
in the hopes that you would answer more
honestly.  Thanks again.

false
```

That was a pretty long program, with lots of repetition. (All of the sections of code around the questions about

Mexican food were identical, and the bed-wetting question was only slightly different.) Repetition is a bad thing. Still, we can't make it into a big loop or iterator, because sometimes we have things we want to do between questions. In situations like these, it's best to write a method. Here's how:

```
def sayMoo
  puts 'mooooooo...'
end
```

Uh... our program didn't sayMoo. Why not? Because we didn't tell it to. We told it how to sayMoo, but we never actually said to do it. Let's give it another shot:

```
def sayMoo
  puts 'mooooooo...'
end

sayMoo
sayMoo
puts 'coin-coin'
sayMoo
sayMoo
```

```
mooooooo...
mooooooo...
coin-coin
mooooooo...
mooooooo...
```

Ahhh, much better. (Just in case you don't speak French, that was a French duck in the middle of the program. In France, ducks say "coin-coin".)

So we defined the method sayMoo. (Method names, like variable names, start with a lowercase letter. There are a few exceptions, though, like + or ==.) But don't methods always have to be associated with objects? Well, yes they do, and in this case (as with puts and gets), the method is just associated with the object representing the whole program. In the next chapter we'll see how to add methods to other objects. But first...

Method Parameters

You may have noticed that some methods (like gets, to_s, reverse...) you can just call on an object. However, other methods (like +, -, puts...) take parameters to tell the object how to do the method. For example, you wouldn't just say 5+, right? You're telling 5 to add, but you aren't telling it what to add.

To add a parameter to sayMoo (let's say, the number of moos), we would do this:

```
def sayMoo numberOfMoos
  puts 'mooooooo...'*numberOfMoos
end
```

```
sayMoo 3
puts 'oink-oink'
sayMoo  # This should give an error because the parameter is missing.
```

```
mooooooo...mooooooo...mooooooo...
oink-oink
#<ArgumentError: wrong number of arguments (given 0, expected 1)>
```

numberOfMoos is a variable which points to the parameter passed in. I'll say that again, but it's a little confusing: numberOfMoos is a variable which points to the parameter passed in. So if I type in sayMoo 3, then the parameter is 3, and the variable numberOfMoos points to 3.

As you can see, the parameter is now required. After all, what is sayMoo supposed to multiply 'mooooooo...' by if you don't give it a parameter? Your poor computer has no idea.

If objects in Ruby are like nouns in English, and methods are like verbs, then you can think of parameters as adverbs (like with sayMoo, where the parameter told us how to sayMoo) or sometimes as direct objects (like with puts, where the parameter is what gets putsed).

Local Variables

In the following program, there are two variables:

```
def doubleThis num
  numTimes2 = num*2
  puts num.to_s+' doubled is '+numTimes2.to_s
end

doubleThis 44
```

```
44 doubled is 88
```

The variables are num and numTimes2. They both sit inside the method doubleThis. These (and all of the variables you have seen so far) are local variables. This means that they live inside the method, and they cannot leave. If you try, you will get an error:

```
def doubleThis num
  numTimes2 = num*2
  puts num.to_s+' doubled is '+numTimes2.to_s
end

doubleThis 44
puts numTimes2.to_s
```

```
44 doubled is 88
#<NameError: undefined local variable or method `numTimes2' for #
<StringIO:0x00000002e78130>>
```

Undefined local variable... In fact, we did define that local variable, but it isn't local to where we tried to use it; it's local to the method.

This might seem inconvenient, but it actually quite nice. While it does mean that you have no access to variables inside methods, it also means that they have no access to your variables, and thus can't screw them up:

```ruby
def littlePest var
  var = nil
  puts 'HAHA!  I ruined your variable!'
end

var = 'You can\'t even touch my variable!'
littlePest var
puts var
```

```
HAHA!  I ruined your variable!
You can't even touch my variable!
```

There are actually two variables in that little program named var: one inside littlePest, and one outside of it. When we called littlePest var, we really just passed the string from one var to the other, so that both were pointing to the same string. Then littlePest pointed its own local var to nil, but that did nothing to the var outside the method.

Return Values

You may have noticed that some methods give you something back when you call them. For example, gets returns a string (the string you typed in), and the + method in 5+3, (which is really 5.+(3)) returns 8. The arithmetic methods for numbers return numbers, and the arithmetic methods for strings return strings.

It's important to understand the difference between methods returning a value to where the method was called, and your program outputting information to your screen, like puts does. Notice that 5+3 returns 8; it does not output 8.

So what does puts return? We never cared before, but let's look at it now:

```ruby
returnVal = puts 'This puts returned:'
puts returnVal
```

```
This puts returned:
```

The first puts didn't seem to return anything, and in a way it didn't; it returned nil. Though we didn't test it, the second puts did, too; puts always returns nil. Every method has to return something, even if it's just nil.

Take a quick break and write a program to find out what sayMoo returned.

Were you surprised? Well, here's how it works: the value returned from a method is simply the last line of the method. In the case of sayMoo, this means it returns puts 'mooooooo...'*numberOfMoos, which is just nil since puts always returns nil. If we wanted all of our methods to return the string 'yellow submarine', we would just need to put that at the end of them:

```
def sayMoo numberOfMoos
  puts 'mooooooo...'*numberOfMoos
  'yellow submarine'
end

x = sayMoo 2
puts x
```

```
mooooooo...mooooooo...
yellow submarine
```

So, let's try that psychology experiment again, but this time we'll write a method to ask the questions for us. It will need to take the question as a parameter, and return true if they answered yes and false if they answered no. (Even though most of the time we just ignore the answer, it's still a good idea for our method to return the answer. This way we can use it for the bed-wetting question, too.) I'm also going to shorten the greeting and the debriefing, just so this is easier to read:

```
def ask question
  goodAnswer = false
  while (not goodAnswer)
    puts question
    reply = gets.chomp.downcase

    if (reply == 'yes' or reply == 'no')
      goodAnswer = true
      if reply == 'yes'
        answer = true
      else
        answer = false
      end
    else
      puts 'Please answer "yes" or "no".'
    end
  end

  answer  # This is what we return (true or false).
```

```
end

puts 'Hello, and thank you for...'
puts

ask 'Do you like eating tacos?'      # We ignore this return value.
ask 'Do you like eating burritos?'
wetsBed = ask 'Do you wet the bed?'  # We save this return value.
ask 'Do you like eating chimichangas?'
ask 'Do you like eating sopapillas?'
ask 'Do you like eating tamales?'
puts 'Just a few more questions...'
ask 'Do you like drinking horchata?'
ask 'Do you like eating flautas?'

puts
puts 'DEBRIEFING:'
puts 'Thank you for...'
puts
puts wetsBed
```

```
Hello, and thank you for...

Do you like eating tacos?
yes
Do you like eating burritos?
yes
Do you wet the bed?
no way!
Please answer "yes" or "no".
Do you wet the bed?
NO
Do you like eating chimichangas?
yes
Do you like eating sopapillas?
yes
Do you like eating tamales?
yes
Just a few more questions...
Do you like drinking horchata?
yes
Do you like eating flautas?
yes
```

```
DEBRIEFING:
Thank you for...

false
```

Not bad, huh? We were able to add more questions (and adding questions is easy now), but our program is still quite a bit shorter! It's a big improvement — a lazy programmer's dream.

One More Big Example

I think another example method would be helpful here. We'll call this one englishNumber. 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 only work on integers from 0 to 100.

(NOTE: This method uses a new trick to return from a method early using the return keyword, and introduces a new twist on branching: elsif. It should be clear in context how these work.)

```ruby
def englishNumber number
  # We only want numbers from 0-100.
  if number < 0
    return 'Please enter a number zero or greater.'
  end
  if number > 100
    return 'Please enter a number 100 or lesser.'
  end

  numString = ''  # 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 to write out?
  left  = left - write*100  # Subtract off those hundreds.

  if write > 0
    return 'one hundred'
  end

  write = left/10          # How many tens left to write out?
  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
      numString = numString + 'ten'
    elsif left == 1
      numString = numString + 'eleven'
    elsif left == 2
      numString = numString + 'twelve'
    elsif left == 3
      numString = numString + 'thirteen'
    elsif left == 4
      numString = numString + 'fourteen'
    elsif left == 5
      numString = numString + 'fifteen'
    elsif left == 6
      numString = numString + 'sixteen'
    elsif left == 7
      numString = numString + 'seventeen'
    elsif left == 8
      numString = numString + 'eighteen'
    elsif left == 9
      numString = numString + 'nineteen'
    end
    # Since we took care of the digit in the ones place already,
    # we have nothing left to write.
    left = 0
  elsif write == 2
    numString = numString + 'twenty'
  elsif write == 3
    numString = numString + 'thirty'
  elsif write == 4
    numString = numString + 'forty'
  elsif write == 5
    numString = numString + 'fifty'
  elsif write == 6
    numString = numString + 'sixty'
  elsif write == 7
    numString = numString + 'seventy'
  elsif write == 8
    numString = numString + 'eighty'
  elsif write == 9
    numString = numString + 'ninety'
  end

  if left > 0
    numString = numString + '-'
  end
```

```ruby
  end

  write = left  # How many ones left to write out?
  left  = 0     # Subtract off those ones.

  if write > 0
    if    write == 1
      numString = numString + 'one'
    elsif write == 2
      numString = numString + 'two'
    elsif write == 3
      numString = numString + 'three'
    elsif write == 4
      numString = numString + 'four'
    elsif write == 5
      numString = numString + 'five'
    elsif write == 6
      numString = numString + 'six'
    elsif write == 7
      numString = numString + 'seven'
    elsif write == 8
      numString = numString + 'eight'
    elsif write == 9
      numString = numString + 'nine'
    end
  end

  if numString == ''
    # The only way "numString" could be empty is if
    # "number" is 0.
    return 'zero'
  end

  # If we got this far, then we had a number somewhere
  # in between 0 and 100, so we need to return "numString".
  numString
end

puts englishNumber(  0)
puts englishNumber(  9)
puts englishNumber( 10)
puts englishNumber( 11)
puts englishNumber( 17)
puts englishNumber( 32)
puts englishNumber( 88)
puts englishNumber( 99)
puts englishNumber(100)
```

```
zero
nine
ten
eleven
seventeen
thirty-two
eighty-eight
ninety-nine
one hundred
```

Well, there are certainly a few things about this program I don't like. First, it has too much repetition. Second, it doesn't handle numbers greater than 100. Third, there are too many special cases, too many returns. Let's use some arrays and try to clean it up a bit:

```ruby
def englishNumber number
  if number < 0  # No negative numbers.
    return 'Please enter a number that isn\'t negative.'
  end
  if number == 0
    return 'zero'
  end

  # No more special cases! No more returns!

  numString = ''  # This is the string we will return.

  onesPlace = ['one',     'two',       'three',    'four',     'five',
               'six',     'seven',     'eight',    'nine']
  tensPlace = ['ten',     'twenty',    'thirty',   'forty',    'fifty',
               'sixty',   'seventy',   'eighty',   'ninety']
  teenagers = ['eleven',  'twelve',    'thirteen', 'fourteen', 'fifteen',
               'sixteen', 'seventeen', 'eighteen', 'nineteen']

  # "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 to write out?
  left  = left - write*100  # Subtract off those hundreds.

  if write > 0
```

```
    # Now here's a really sly trick:
    hundreds  = englishNumber write
    numString = numString + hundreds + ' hundred'
    # That's called "recursion". So what did I just do?
    # I told this method to call itself, but with "write" instead of
    # "number". Remember that "write" is (at the moment) the number of
    # hundreds we have to write out. After we add "hundreds" to
    # "numString", we add the string ' hundred' after it.
    # So, for example, if we originally called englishNumber with
    # 1999 (so "number" = 1999), then at this point "write" would
    # be 19, and "left" would be 99. The laziest thing to do at this
    # point is to have englishNumber write out the 'nineteen' for us,
    # then we write out ' hundred', and then the rest of
    # englishNumber writes out 'ninety-nine'.

    if left > 0
      # So we don't write 'two hundredfifty-one'...
      numString = numString + ' '
    end
  end

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

  if write > 0
    if ((write == 1) and (left > 0))
      # Since we can't write "tenty-two" instead of "twelve",
      # we have to make a special exception for these.
      numString = numString + teenagers[left-1]
      # The "-1" is because teenagers[3] is 'fourteen', not 'thirteen'.

      # Since we took care of the digit in the ones place already,
      # we have nothing left to write.
      left = 0
    else
      numString = numString + tensPlace[write-1]
      # The "-1" is because tensPlace[3] is 'forty', not 'thirty'.
    end

    if left > 0
      # So we don't write 'sixtyfour'...
      numString = numString + '-'
    end
  end
  write = left  # How many ones left to write out?
  left  = 0     # Subtract off those ones.

  if write > 0
    numString = numString + onesPlace[write-1]
    # The "-1" is because onesPlace[3] is 'four', not 'three'.
  end

  # Now we just return "numString"...
  numString
end

puts englishNumber(  0)
puts englishNumber(  9)
puts englishNumber( 10)
puts englishNumber( 11)
puts englishNumber( 17)
puts englishNumber( 32)
puts englishNumber( 88)
puts englishNumber( 99)
puts englishNumber(100)
puts englishNumber(101)
puts englishNumber(234)
puts englishNumber(3211)
puts englishNumber(999999)
puts englishNumber(1000000000000)
```

```
zero
nine
ten
eleven
seventeen
thirty-two
eighty-eight
ninety-nine
one hundred
one hundred one
two hundred thirty-four
thirty-two hundred eleven
ninety-nine hundred ninety-nine hundred ninety-nine
one hundred hundred hundred hundred hundred hundred
```

Ahhhh.... That's much, much better. The program is fairly dense, which is why I put in so many comments. It even works for large numbers... though not quite as nicely as one would hope. For example, I think 'one trillion' would be a nicer return value for that last number, or even 'one million million' (though all three are correct). In fact, you can do that right now...

A Few Things to Try

Expand upon englishNumber. First, put in thousands. So it should return 'one thousand' instead of 'ten hundred' and 'ten thousand' instead of 'one hundred hundred'.

Expand upon englishNumber some more. Now put in millions, so you get 'one million' instead of 'one thousand thousand'. Then try adding billions and trillions. How high can you go?

How about weddingNumber? It should work almost the same as englishNumber, except that it should insert the word "and" all over the place, returning things like 'nineteen hundred and seventy and two', or however wedding invitations are supposed to look. I'd give you more examples, but I don't fully understand it myself. You might need to contact a wedding coordinator to help you.

"Ninety-nine bottles of beer..." Using englishNumber and your old program, write out the lyrics to this song the right way this time. Punish your computer: have it start at 9999. (Don't pick a number too large, though, because writing all of that to the screen takes your computer quite a while. A hundred thousand bottles of beer takes some time; and if you pick a million, you'll be punishing yourself as well!

Congratulations! At this point, you are a true programmer! You have learned everything you need to build huge programs from scratch. If you have ideas for programs you would like to write for yourself, give them a shot!

Of course, building everything from scratch can be a pretty slow process. Why spend time writing code that someone else already wrote? Would you like your program to send some email? Would you like to save and load files on your computer? How about generating web pages for a tutorial where the code samples are all automatically tested? 😉 Ruby has many different kinds of objects we can use to help us write better programs faster.