Source File Layout

Here is how we might structure the source file for a program with several methods:

```
def main
   puts "in main"; f; g
end

def f; puts "in f" end

def g; puts "in g" end

main
```

Note that even though main calls f and g, the definitions of those methods do *not* need to precede main.

What actually runs the program is the line "main" at the end. It *is* required that the definition of main has been seen by then. And, because main uses f and g, they need to have been seen, too.

Question: At the time the definition of main is processed, what's known about f and g?

Try shuffling around the three definitions and "main" to see what works and what doesn't.

Blocks in Ruby vs. anonymous functions in ML

Here is a Ruby iterator that yields the first and last characters in a string:

```
def f_L(s)
   yield s[0,1]
   yield s[-1,1]
end

f_L("abc") { |c| puts c }
```

Here is a rough analog in ML:

```
fun f_L s f = (
   f(String.sub(s,0));
   f(String.sub(s,size(s) - 1)));

f_L "abc" (fn(s) => print(str(s)^"\n"))
```

In both languages, a chunk of code is twice handed a value and the code is executed.

One way to think of 'yield x' is 'call_associated_block(x)'.

Blocks vs. anonymous functions, continued

Here is map in ML:

```
fun map F [ ] = [ ]
 |  map F (x::xs) = F(x)::(map F xs)

- map (fn(n) => n * 2) [10,20,30];
val it = [20,40,60] : int list
```

Here is map in Ruby:

```
def map(a)
   map_result = [ ]
   for x in a do
      block_result = yield x
      map_result << block_result
   end
   map_result
end

>> map([10,"20",[30]]) { |n| n * 2 }
=> [20, "2020", [30, 30]]
```

gets: not just for standard input

The slides show a freestanding call to gets (which invokes Kernel#gets) as a way to read a line from standard input. In fact, the behavior of gets is fairly complex.

Here's a slightly abridged excerpt from the documentation on gets:

> Returns the next line from the list of files in ARGV, or from standard input if no files are present on the command line.

Through experimentation, here's what I've found:

> If gets is called for the first time and ARGV is not empty, then File.open(ARGV[0]) is called and the first element of ARGV is removed. Let's say that the result, an instance of File, is assigned to f. Then f.gets is called and the result of that becomes the result of gets. Each subsequent call to gets results in a call to f.gets. This continues until end of file is reached on f. This process repeats until ARGV is empty.

In other words, if files are named on the command line, then a loop like "while line = gets" will produce each line of each file.

gets, continued

We can take advantage of this behavior of gets to produce a trivial implementation of an option-less version of cat:

```
while line = gets do
   puts line
end
```

Usage:

```
% wc -l a b c
 2 a
 4 b
 1 c
 7 total
% ruby cat.rb a b c | wc -l
7
```

Options, like -n and -v could be accommodated by scanning ARGV and removing them before loop, leaving only the files to open in ARGV.

gets, continued

Question: Is it a good idea to provide this elaborate but often convenient behavior?

My first-ever run of a Ruby program that processed options and read standard input with gets looked like this:

```
% ruby testopts.rb -x -v 1 < x
testopts.rb: in `gets': No such file or directory - -x (Errno::ENOENT)
```

It took me a little while to figure out what was going on. (Do you see the problem?)

There's the "pro" of convenience with gets, but there are some "cons". Here are a few:

- For the occasional Ruby user this behavior is one more thing to remember.
- A Ruby instructor focused on minimal but sufficient coverage must decide whether to even mention the freestanding form or simply teach STDIN.gets.
- A person documenting the library would need to write a fair amount to completely describe this behavior and disposition of the various errors that may occur.
- A suite of tests for the library would have to have quite a few tests to cover gets.