The REPL

IDIOMS
`call`: a method that means "do that thing you do"
CWD: an acronym for "Current Working Directory", it's what `pwd` prints

FILES
`$LOAD_PATH`: stores an array of directories that will be searched sequentially when you require a file`
`File.expand_path("c/d", "/a/b")` expands to "/a/b/c/d"
`__dir__`: the path to the directory of the current file relative to CWD
`__FILE__`: the path to the current file relative to CWD

BINARY
definition: The file you actually run
responsibility - Wire up the world and kick things off
The binary is usually invoked directly (`$ ./numbermind`), though not in our Numbermind example (`ruby numbermind.rb`)
Why "binary"? a long time ago, executable files were always machine code, so 1s and 0s,
which is called "binary", because there are only 2 values for each digit.

CLI
Command Line Interface - code that connects a user on the command-line to the code

IO (Input and Output)
`$stdin`: "standard input", the text input to our program
`$stdout`: "standard output", the text output of our program
`gets`: shortcut for `$stdin.gets`
`puts`: shortcut for `$stdout.puts`
<u>Beware!</u>
text read from `$stdin` comes from the world outside our program, we don't control it
text written to `$stdout` goes to the world outside our program, we don't control it
this means that if we can't tell our code what to read and write from, it's not testable

Common patten
Instead of talking to the global variables, let the caller pass us the stream to talk to.
This lets us pass it a stream with input we've selected for tests

REPL
Read: get user input
Eval: process it in some manner
Print: prints results
Loop: repeat these steps

StringIO

An IO object, like our $stdin and $stdout
but reads from a string instead of the standard input
and writes to a string instead of the standard output

```
# has hidden dependency on standard output (monitor)
def print_greeting
  puts "Hello!"
end
print_greeting # goes to monitor

# lets us choose where it prints the greeting
def print_greeting(stream)
  stream.puts "Hello!"
end

# choose stdout -- goes to monitor
print_greeting $stdout

# choose a different IO object -- does not affect the outside world
require 'stringio'
stream = StringIO.new
print_greeting stream
stream.string # => "Hello!\n"
```

Example: Calculator REPL

```
def calculator_repl(instream, outstream, calcualted)
  outstream.puts "The current number is: #{calcualted}"
  outstream.puts "Enter an operator and number, e.g. '+5', or 'q' to quit"
  loop do
    # prompt and get input
    outstream.print "> "
    raw_input = instream.gets.strip

    # potentially quit calculating, returning the calculated value
    return calcualted if raw_input == 'q'

    # parse the input for an operator or sequence of digits
    inputs   = raw_input.scan(/[-+*\/]|\d+/) # in "+5", this is ["+", "5"]
    operator = inputs[0]
    number   = inputs[1].to_f

    # perform the calculation
    if    operator == '+' then result = calcualted +  number
    elsif operator == '-' then result = calcualted -  number
    elsif operator == '*' then result = calcualted *  number
    elsif operator == '/' then result = calcualted /  number
    end

    # show the calculation, update the calculated value
    outstream.puts("#{calcualted} #{operator} #{number} = #{result}")
    calcualted = result
  end
end

# Read input from our object's stream, not the $stdin, write to $stdout
require 'stringio'
instream = StringIO.new("+2\n *3\n -4\n q\n")
calculator_repl(instream, $stdout, 0.0) # => 2.0

# >> The current number is: 0.0
# >> Enter an operator and number, e.g. '+5', or 'q' to quit
# >> > 0.0 + 2.0 = 2.0
# >> > 2.0 * 3.0 = 6.0
# >> > 6.0 - 4.0 = 2.0
# >> >
```