Lab: Recursion in JavaTown

In this lab, we will explore methods that call other methods of the same object, and even recursive methods—ones that call themselves. Here's a simple example of a class in which one method calls another.

```
public class Squarer
{
   private square(x)
   {
      return x * x;
   }

   public squareSum(x, y)
   {
      return this.square(x + y);
   }
}
```

Notice that the author of this class is forbidding other classes from calling the `square` method, by designating it as `private`.

Here's an example of a method that calls itself.

```
public fact(n)
{
   if (n == 0)
      return 1;                      the base case
   return n * this.fact(n - 1);    the recursive case
}
```

In this lab, you will write a number of recursive methods. Be sure to test that each one works correctly before moving on to the next exercise.

Exercise: A Power Trip

Create a new text file. In it complete the `FancyCalc` class definition shown below, so that the `pow` method correctly computes *base*^{exponent}. You may find it helpful to think about how knowing the value of *base*$^{exponent - 1}$ could help you find *base*^{exponent}.

```
public class FancyCalc
{
   public pow(base, exponent)
   {
      if (exponent == 0)
         return __________;
      else
         return __________;
   }
}
```

It will take pow 10 seconds since the big O is linear, which means 2n = 2t

Food for Thought: If it takes `pow` 5 seconds to run when `exponent` is 1000, about how long will it take to run when `exponent` is 2000?

Exercise: Super Powers

Complete the `fastPow` method below, add add it to your `FancyCalc` class. The `fastPow` method should also compute *base*^{exponent}. (Be sure to write a `square` method, too!)

```
public fastPow(base, exponent)
{
   //base case
   if (exponent == 0)
      return __________;

   if (exponent % 2 == 0)
      return this.square(
                  this.fastPow(__________, __________));

   return base * this.fastPow(__________, __________);
}
```

Food for Thought: If it takes `fastPow` 5 seconds to run when `exponent` is 1000, about how long will it take to run when `exponent` is 2000?

Fast pow will take a bit more than 5 seconds. Because 2000 is just 2 * 1000, fast pow will only take 1 more step and then its just 1000 again, so a bit over 5 seconds

Exercise: Remainder Danger

Euclid's algorithm tells us how to compute the greatest common divisor (GCD) of two positive integers *a* and *b*. Using Euclid's algorithm, to find the GCD of 206 and 40 (for example), first find the remainder when 206 is divided by 40. Then find the GCD of 40 and this remainder (which turns out to be 6), using the same idea again. When you reach the point where the second number is 0, the first number will be the GCD of 206 and 40 that you were looking for, as shown below.

$$
\begin{aligned}
&gcd(206, 40) \\
&\quad = gcd(40, 6) \\
&\quad = gcd(6, 4) \\
&\quad = gcd(4, 2) \\
&\quad = gcd(2, 0) \\
&\quad = 2
\end{aligned}
$$

Add a method `gcd` to your `FancyCalc` class. This method should use Euclid's algorithm to return the greatest common divisor of two positive integers.

Exercise: Prime Suspect

Add the method `isPrime` and its helper `helpPrime` to your `FancyCalc` class. Your completed `isPrime` method will return the boolean value *`true`* when its input is a prime number, and *`false`* otherwise. The `isPrime` method tests if `num` is prime by trying to divide it by all integers from 2 to `num` – 1. For example, to test if the number 7 is prime, it will try dividing 7 by 2, 3, 4, 5, and 6, before declaring that 7 is indeed prime. On the other hand, to test if the number 9 is prime, this method will try dividing 9 by 2, and then 3, before declaring that 9 is not prime.

```
public isPrime(num)
{
   return this.helpPrime(num, 2);
}

private helpPrime(num, divisor)
{
   if (divisor == num)
      return __________;

   if (num % divisor == 0)
      return __________;

   return this.helpPrime(__________, __________);
}
```

Exercise: Just the Facts

Complete the `fact` and `factHelp` methods shown below, and add them to your `FancyCalc` class. The `fact` method should correctly compute the factorial function. (Note that this is *not* the same way we computed factorial in class.)

```
public fact(n)
{
   return this.factHelp(n, __________);
}

private factHelp(n, result)
{
   if (n == 0)
      return result;
   else
      return this.factHelp(n - 1, __________);
}
```

Exercise: Telling Fibs

The n^{th} element of the Fibonacci sequence (0, 1, 1, 2, 3, 5, 8, 13, ...) can be defined as follows:

$$fib(0) = 0$$
$$fib(1) = 1$$
$$fib(n) = fib(n-1) + fib(n-2)$$

Add a method to your `FancyCalc` class called `fib`, which should use the above recursive definition to return the `n`th Fibonacci number.

Food for Thought: If it takes `fib` 5 seconds to run when `n` is 100, about how long will it take to run when `n` is 101?

Fib will take 10 seconds because n+1 means 2t, since
it will run fib(n-1) one more time which is a whole tree from the top so double.

The Root of the Problem

In math, the following would suffice to tell us *what is* a square-root.

$\sqrt{x}$ = the y such that $y \geq 0$ and $y^2 = x$

To program the computer to find square-roots for us, though, we'll need a strategy that tells the computer *how to* compute a square-root. In this part of the lab, you'll use Newton's method to implement a square-root method, in which the computer repeatedly makes a guess and adjusts it to make a better guess.

Here's how Newton's method computes square-roots. Let's say we want to compute $\sqrt{43}$. We start by guessing 43 and calculating 43 / *guess*. Then we average these, and this tells us our next guess, as shown in the following table. (Note that JavaTown only supports integer division, so all decimal places are dropped.)

guess	*num / guess*	average of *guess* and *num / guess*
43	1	22
22	1	11
11	3	7
7	6	6
6	7	6

The algorithm stops when either *guess* is equal to the next guess, or when the next guess would begin to increase. At this point, *guess* is the answer. (Technically, we're computing the greatest integer whose square is no larger than our number. This may strike you as a lot of work for a not-so-satisfying result, but the situation is actually pretty good. First, we can compute square-roots of enormous numbers quite quickly with this algorithm. Second, we can compute decimal places by using large numbers. To find the square-root of x with d places after the decimal point, we append $2d$ zeroes to x. For example, to find the square-root of 2 with 3 decimal places, we run the algorithm on 2,000,000. The answer 1414 tells us that the square-root of 2 is actually near 1.414.)

Try filling in the following table for computing $\sqrt{81}$.

guess	*num / guess*	average of *guess* and *num / guess*
81	1	41
41	1	21
21	3	12
12	6	9
9	9	9

Try filling in the following table for computing √15. What do you notice?

guess	*num / guess*	average of *guess* and *num / guess*
15	1	8
8	1	4
4	3	3
3	5	4

After the average got
too small
it started
to increase,
which is when we
stop

When you understand the algorithm, add a method called `sqrt` to your `FancyCalc` class. This method should take a single argument (a positive number) and return its square-root using the algorithm described above. You should break this task down into a number of helper methods, which may include (and is certainly not limited to) the following:

- a method that returns the average of two numbers.
- a method that returns true when the algorithm should stop (when *guess* is close enough to *num* / *guess*).
- a method that takes in a guess and a number, and uses Newton's method to improve that guess until it finds and returns the square root of the number.

Other Additional Opportunities

- Time how long the recursive methods take for different inputs, and identify patterns.
- Write a method that takes an `Enumerator` as input and returns the sum of the values that the `Enumerator` returns. You need an `Enumerator` class. Use the `Enumerator` from the previous lab.
- Write a method that runs much slower than `fib`. (Not just twice as slow or 100 times as slow, etc. Hint: Look up Ackermann's function.)
- Write `fastFib`, which finds the n^{th} Fibonacci number *much* faster than `fib`.
- Write a terminating recursive method that doesn't use an `if` statement.
- Improve your `sqrt` stopping condition to handle the square roots of 3, 8, 15, 24, etc., correctly.

How to Get Checked Off

You must show me the file in which you have defined the `FancyCalc` class. In addition, you must demonstrate that its methods work properly. You should simply show all this to me on your computer, during class or during extra help. You must also upload your files.