1 Adder

Leo, a diligent CS61B student, was trying to submit HW1 when his computer crashed. Upon rebooting, he discovered that some portions of his code had disappeared! Help Leo fix his homework so that he can turn it in on time.

```
public class Adder {

    /** Adds the range of numbers from 1 to N recursively.
     *  @param N range of numbers being added
     *  @return  sum of numbers in range
     */
    public static int addRange(int n) {
        /* Base case */
        if (n == 1) {
                return 1;
        }

        /* Recurse! */
        return n + addRange(n - 1);
    }
}
```

Let's first look at `public static int addRange(int n)`. This is called the *method header*.

- The `int` in the method header refers to the return type of this function. This lets the compiler as well as other developers know that the `addRange` function returns an integer.
- The `(int n)` after the method name refers to the parameters that the method accepts. The `addRange` method takes in one integer `n`.
- You will learn what `public` and `static` mean later in this course.

The *method signature* is `addRange(int n)`. No two methods in the same class can have the same method signature.

The solution to this problem relies on *recursion*. Recall from CS 61A that a function is called *recursive* if the body of that function calls itself, either directly or indirectly.
In this case, we decrement the argument to each recursive call to `addRange` each time till we reach the base case of `n == 1`.

2 Fibonacci Numbers

The next problem took Leo several days, but he only has an hour until his homework is due. Help him out!

```java
/** The Fibonacci sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, ... */
public class Fibonacci {

    /** fib1(N) is the N^th Fibonacci number, for N ≥ 0. fib1(N)
     * is tree recursive. */
    public static int fib1(int n) {
        if (n <= 1) {
            return n;
        }
        return fib(n - 1) + fib(n - 2);
    }
```

This method is *tree recursive* because it calls itself more than once within its body. This solution turns out to be quite inefficient because its branching factor leads to exponential growth.

For example, `fib(n-1)` will call `fib(n-2)` and `fib(n-3)`. Thus, we will be calling `fib(n-2)` twice, and `fib(n-3)` three times, and so on. How many times is `fib(n-4)` called? Hint, it's not four.

```java
    /** fib2(N, K, F0, F1) is the N^th Fibonacci number, assuming
     * that F0 and F1 are the K-1^th and K^th Fibonacci numbers,
     * 1 ≤ K ≤ N. Thus, fib2(N, 1, 0, 1) is simply the N^th
     * Fibonacci number. */
    public static int fib2(int n, int k, int f0, int f1) {
        if (k == n) {
            return f1;
        } else {
            return fib2(n, k + 1, f1, f1 + f0);
        }
    }
```

As opposed to `fib1`, `fib2` is *tail-recursive*. This means that we don't have to recompute already known values.

```java
/** fib3(N) is the N^th Fibonacci number, for N ≥ 0. fib3(N)
 *  is iterative. */
public static int fib3(int n) {
    if (n == 0) {
        return 0;
    }
    int prev = 0;
    int curr = 1;
    for (int i = 1; i < n; i += 1) {
        int tmp = curr;
        curr = prev + curr;
        prev = tmp;
    }
    return curr;
}
```

Nothing special here. This is just an iterative version of `fib`.

}