Answers to Self-Study Questions

Test Yourself #1

The `printInt` method does obey recursion rule 1; it does have a base case, namely:

```
if (k == 0) return;
```

It obeys recursion rule 2 in a sense; the code:

```
printInt( k - 1 );
```

makes progress toward the base case as long as `k` is greater than zero. However, all calls with actual parameters ***less*** than zero will cause an infinite recursion. Assuming that the intention is to print only positive values, this could be fixed by changing the base case to:

```
if (k <= 0)
    return;
```

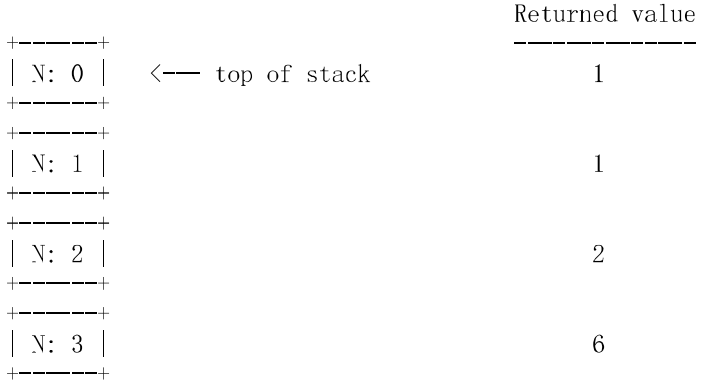
Test Yourself #2

The call `printTwoInts(3)` causes the following to be printed:

```
From before recursion: 3
From before recursion: 2
From before recursion: 1
From after recursion: 1
From after recursion: 2
From after recursion: 3
```

Test Yourself #3

Question 1: The call `factorial(3)` leads to three recursive calls; when all calls are still active, the runtime stack would be as shown below (showing just the values of parameter N). The values returned by each call are also shown.



Question 2: The iterative version of factorial will return -1 as the result of the call `factorial(-1)`. The recursive version will go into an infinite recursion (because the base case, N==0, will never be reached).

Since, mathematically, factorial is undefined for negative numbers, a call to factorial with a negative number should cause an exception. This is easily done for the iterative version:

```
int factorial(int N) {
    if (N < 0) {
        throw new NegativeValueException();
    }
    if (N == 0) {
        return 1;
```

```
        }
        int tmp = 1;
        for (int k = N; k > 1; k--) {
            tmp = tmp*k;
        }
        return (tmp);
    }
```

(where `NegativeValueException` would have to be defined as a public exception).

For the recursive version, we could change the method to:

```
    int factorial(int N) {
        if (N < 0){
            throw new NegativeValueException();
        }
        if (N == 0) {
            return 1;
        }
        else {
            return (N * factorial( N - 1 ));
        }
    }
```

However, the problem with this is that the check for N < 0 will be made on ***every*** call, including the recursive calls where it cannot be true. This makes the method slightly less efficient. A better solution would be to make the check once, the first time factorial is called and then use an auxiliary recusive method with no check:

```
    int factorial(int N) {
        if (N < 0) {
            throw new NegativeValueException();
        }
        else {
            return factAux( N );
        }
    }

    int factAux(int N) {
        if (N == 0) {
            return 1;
        }
        else {
            return (N * factAux( N - 1 ));
        }
    }
```

Test Yourself #4

Question 1:

```
    public static int sum(Listnode<Integer> node) {
        if (node == null) {
            return 0;
        }
        int val = node.getData();
        return val + sum(node.getNext());
    }
```

Question 2:

```
    public static String vowels(String str) {
        // Base case
        if (str.length() == 0) {
            return "";
        }
```

```
        String ch = str.substring(0, 1);

        if (ch.equals("a") || ch.equals("e") || ch.equals("i")
                           || ch.equals("o") || ch.equals("u")) {
            return ch + vowels(str.substring(1));
        } else {
            return vowels(str.substring(1));
        }
    }
```

Test Yourself #5

N	T(N)
1	1
2	3
4	7
8	15

The solution is: T(N) = 2*N-1

Here's the verification:

T(N)	=	1 + 2 * T(N/2)	// given
2*N-1	=? =	1 + 2 * (2 * (N/2) - 1)	// replace T(...) on both sides with the guessed solution in which each N is replaced by ...
2*N-1	=? =	1 + 2 * (N - 1)	// simplify
2*N-1	=? =	1 + 2*N - 2	// simplify more
2*N-1	=	2 * N - 1	// yes the two sides are equal!