The assignment 5 did not have any recursion on it. There will be some recursion on your final exam. These questions are designed to give you practice with this.

1. Assume I have the following method:

```
public static void recursiveStars(int n) {
         if (n == 0) {
                return;
         }

         System.out.println("*");
         recursiveStars(n-1);
}
```

   What will be the output if I call this method in the following ways?

   - recursiveStars(0);
   - recursiveStars(1);
   - recursiveStars(2);
   - recursiveStars(10);
   - recursiveStars(-10);

2. Consider the following method:

```
public static int doubleRecursion(int n) {
     if (n == 0) return 1;

     System.out.println("Hi");
     return doubleRecursion(n-1) + doubleRecursion(n-1);
}
```

   If I call the method by writing `doubleRecursion(1)` , how many times will the word Hi be printed?

   What about `doubleRecursion(2)`

   What about `doubleRecursion(5)` , how many times will the word Hi be printed?

   What about `doubleRecursion(10)` or `doubleRecursion(100)`

   How can I rewrite this method so that it still uses recursion but so that Hi is printed many fewer times but with the exact same result?

3. Rewrite the following method using a loop instead of recursion:

```
public static String stringMultiplier(int n, String concat) {
     if (n == 0) {
         return "";
     }

    return concat + stringMultiplier(n-1, concat);
}
```

4. Write a method without using any loop to check if a String consists only of upper case letters. Hint: your method should look like:

   ```
   public static boolean checkIfStringIsUpper(String original, int lookedAtSoFar
   ```

5. Write a method that takes as input a String and using recursion converts all of the lower case letters to upper case and all of the upper case letters to lower case

6. Given a String s, write a method to return an ArrayList¡String¿ representing all anagrams of the String. In otherwords, you should come up with all possible arrangements of the characters in s.

   Hint: Think about how you would do this if s has length 1. Then think about how you would do this if s has length 2.

7.
   ```
   int i1 = 7;
   int i2 = 9;
   double[] a1 = {6.0, 1.0, 3.5, 4.0, 2.0};
   double[] a2 = a1;
   a1[0] = a1[0] + a1[1];
   a2[1] = i1 / i2;
   a2[2] = (int)a2[2] * 3;
   a2[3] *= a2[3] + 1.0;
   a2[4] = 3.0 * (i1 % i2);
   System.out.println("a2[0]: " + a2[0]);
   System.out.println("a2[1]: " + a2[1]);
   System.out.println("a2[2]: " + a2[2]);
   System.out.println("a2[3]: " + a2[3]);
   System.out.println("a2[4]: " + a2[4]);
   ```

   Answer:

   ```
   7.0
   0.0
   9.0
   20
   21
   ```

   Here there are a couple tricks. For a2[0], the key thing is that a2 and a1 are aliases of the same array. This is the case since we say `double[] a2 = a1` *which means "store the value of a1 into the variable a2" Since a1 stores an address as it's value, this means a2 and a1 will both store the same address. Thus when we change a1[0], we are also changing a2[0]*

   For a2[1] we have an integer division. Remember you always round down. Even though we later on store this in a double, the division has been performed first. To keep it as a double, you could cast it, by writing `a2[1] = (double)i1 / i2` *or* `a2[1] = i1 / (double) i2` *but not* `(double) (i1 / i2)` *which would also do the division before casting.*

   For a2[2], the thing to know is that the casting is performed first, before the multiplication. This is, in fact, the reason we can write `a2[1] = (double)i1 / i2` *because even without ( ) the first thing done is the cast*

   *For a2[3] , you have to know what the *= operator does. In general, when I have*

   ```
   variable *= value
   ```

It means `variable = variable * value` *The same is true for the operators +=, -= and /=*

For a2[4], you need to know the modulus % operator. Remember that it is like a clock. Here we have 7 % 9. This means "It is 7 hours after noon on a clock with 9 hours" If that were the case, the current time would be 7.

```
int i1 = 4;
int i2 = 6;
double[] a1 = {7.0, 1.0, 3.5, 4.0, 1.0};
double[] a2 = a1;
a1[0] = a1[0] + a1[1];
a2[1] = i1 / i2;
a2[2] = (int)a2[2] * 3;
a2[3] *= a2[3] + 1.0;
a2[4] = 3.0 * (i1 % i2);
System.out.println("a2[0]: " + a2[0]);
System.out.println("a2[1]: " + a2[1]);
System.out.println("a2[2]: " + a2[2]);
System.out.println("a2[3]: " + a2[3]);
System.out.println("a2[4]: " + a2[4]);
```

Answer: (see above for explanation

```
8.0
0.0
9.0
20
12
```

8.
```
String pig = "catch-22";
String dog = "CATCH-" + 22;
System.out.println("Animals are equal: " + (pig == dog));
System.out.println(("Animals are equal: " + pig) == dog);
System.out.println("Animals are equal: " + pig == dog);
```

Answer:

```
Animals are equal: false
false
false
```

This question tests order of operations. In the first case, we have ( ) around the boolean expression pig==dog. This means that we will first do this. Since pig==dog evaluates to false, I will end up with "Animals are equal" + false with false as a boolean. Any time I have a String plus a boolean, I will convert the boolean to a String. So we print Animals are equal: false

In the second case, I have the parenthesis differently placed. Here we will first add "Animals are equal:" to "catch-22" This gives me the String "Animals are equal:catch-22" Since this String is not equal to "CATCH-22", we will get the output "false" In the final case, you just need to know which order is done first when no parenthesis is given. Plus is done first. (This is why statements like x+3 == 5 make sense to do

Important note: It is not a compile time error to use == when comparing Strings. This just means to compare the 2 addresses of the Strings. In many cases, this will be a logical error, but in some cases you may actually want to compare the address of 2 Strings to detect if they are aliases or not

9. For each of the following conditions, write a **boolean expression** that evaluates to `true` if and only if the condition is true. You are not allowed to use any methods from the Java Platform API (that is, the Java standard library) to answer this question.

   (a) The values stored in variables `m` (which specifies the number of a month, where January is month `1`) and `d` (which specifies the number of a day within a month), both of type `int`, represent a day that is part of the Winter semester at McGill. Note that the winter semester in 2011 starts on January 4th and ends on April 28th (inclusive of both dates)

   Answer:

```
m == 1 && day >= 4 && day <= 31 || m == 2 && day <= 28 && day >= 1
 || m==3 && day <=31 && day >= 1 || m==4 && day <=28 && day >= 1
```

For this we just need to make sure we include all possible cases. m has to be between 1 and 4 to be a valid semester date. For January, it must be between 4 and 31, for February 2 and 28, etc. Since I said you don't have to check if it's a valid date or not, a sufficient answer for full credit was:

```
m == 1 && day >= 4 || m == 2 || m==3 || m == 4 && day <= 28
```

(b) The character stored in variable `c` (of type `char`) is **NOT** a letter (upper-case or lower-case)

Answer:

```
!(c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z' )
```

You can use the unicode values if you want instead of the chars, but this way you don't need to memorize them. What you are testing is whether the character `c` *has a unicode value between a and z or A and Z, and then taking a NOT of the entire expression. Rather than writing the unicode values out, however, you can simply write the character needed here and have Java evaluate the expression for you. When Java evaluates the expression* `c >= 'a'` *for example, it will see that it has a char variable and a char literal. The only way for it to make this comparison then, is to get the unicode value represented by 'a'*

(c) **Exactly ONE** of the values stored in variables `a`, `b` and `c` (all of type `int`) are **EVEN**.

Answer:

```
a \% 2 + b\% 2 + c\% 2 == 2
```

Whenever a number modulo 2 is equal to 0, it is even. If it is odd, then the number modulo 0 equals 1. So if the sum adds up to 2, then there must be 2 odd numbers, meaning there is 1 even number. It was also, of course, correct to write an if statement with a sequence of ORs and ANDs. Also possible is to take advantage of integer division to test if a number is even. For example `a / 2 * 2 == a` *will evaluate to true whenever a is even, but false if a is odd (since if a is odd, you will lose some precision at this point*

10. Consider the following statement, where `p`, `q`, `r`, and `e` are all variables of type `boolean`:

```
e = (!p && !q) || (q && !r) || !p || r;
```

Complete the following truth table by calculating the value of `e` for each possible combinations of values for `p`, `q`, and `r`, and entering the result in the corresponding row of the truth table.

In other words, for every possible value of `p`, `q`, and `r` write whether the overall expression is `true` or `false`

p	q	r	e
false	false	false	**true**
false	false	true	**true**
false	true	false	**true**
false	true	true	**true**
true	false	false	**false**
true	false	true	**true**
true	true	false	**true**
true	true	true	**true**

The simplest way to think about this is to consider the fact that this main statement is just a sequence of a bunch of ORs. As long as at least one OR is true, the whole expression will be true. `!p && !q` *will be true whenever both p and q are false. This tells us that the first row and second row are true. The second entry* `q && !r` *gives us that the 3rd row and the 7th row are true. The 3rd statement gives us the first 4 rows. The last statement gives us rows 2,4,6, and 8. After we do this we see that the only thing left that is false is the 5th row.*

11. Consider the following program:

```
 public class Errors
  public static void main(String[] args) {


    for (int p = 2; p <= 15; p++) {
      int i = (int) (Math.pow(2, p) - 1);

      if (isPrime(i))  {
        System.out.println(p + "\t" + i + " is prime!");
     }
      else System.out.println("Not prime. Reversal\t"
             + reversal(i));
    }
  }

  public static boolean isPrime(int num) {
    if (num % 2 == 0) {
      return true;
    }

    for (int i = 2; i <= num / 2; i--) {
      if (num % i = 0) {
        return false;
      }
    }
    return true;
  }

  public static int reversal(int num) {
    final int result = 0;

    while (num != 0) {
      int lastDigit = num % 10;
      result = result * 10 + lastDigit;
      num = num / 10;
    }
  }
}
```

The above program is designed to check whether numbers x of the form $2^p - 1$ are prime for integer values of p between 2 and 15, inclusive. The program should display the result for each x. For example, for the number 7 it should display:

```
3       7 is prime!
```

If a particular value of x is not prime, the program should also display this value of x in reverse. For example, 15 is not prime, so the program should print 51 along with a message about 15 not being prime, as follows:

```
Not prime. Reversal        51
```

However, there are **5** errors in this program. Find all the errors and list them. For each error you list, you **MUST** provide the **line number** at which the error occurs, the **type** of error (compile-time, run-time, logical), and a **description** of the error. Suggesting a solution is **NOT** necessary.

Note that stylistic errors DO NOT count as errors for this question.

Do not list more than 5 errors, as you will be penalized for every "error" in excess of 5.

This program actually had 6 errors

(a) Line 1: Compile time error , Missing { at start of class

(b) Line 18: Logical error, When `num % 2 == 0` we have an even number. Thus isPrime should return false

(c) Line 21: Logical error. i– . This is not actually an infinite loop because the 2nd step of the loop will make i equal to 1. When this happens `num % i == 0` is true, so the loop would end. Of course.....

(d) Line 22: Compile time error : You must use 2 equals to compare equality

(e) Line 34 : Compile time error : Trying to assign a new value to result, which is declared as final

(f) Line 37: Compile time error: Exiting the method reversal without returning a value.