Homework 2

Note: The complete solution of problems 3, 4, and 5 require writing *functions*. We will learn functions on October 29, and the due date of this homework is about one week later. You can start working on the logic of these programs now, and add the functions "envelope" later.

1. Calculating e^x

One way of approximating e^x, for small values of x, is given by:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots + \frac{x^N}{N!} .$$

$N!$ denotes the factorial of N, i.e. the product of all integers between 1 and N. The above expression is also known as the *MacLaurin polynomial* of degree N (associated to the function e^x). The larger N, the more accurate the approximation.

Write a program (`CalcExp`) that gets two command-line arguments, x and N, and approximates e^x by calculating the MacLaurin polynomial of degree N. Here is an example of the program execution:

```
% java CalcExp 2 10

e raised to the power of 2.0 according to Java: 7.38905609893065
Same, using MacLaurin approximation with 10 steps: 7.388994708994708
```

Tips: (1) the command-line argument x should be parsed and handled as a `double`; (2) Java's benchmark value is obtained by calling the library function `Math.exp(x)`; (3) Watch out for differences between using `int` and using `double`; (4) Start with some fixed values, say $x = 2.0$ and $N = 5$, and worry about the command-line arguments later; (5) Start by writing a loop that outputs the powers and the factorials, for example (for $x = 2.0$ and $N = 5$):

```
2.0 1
4.0 2
8.0 6
16.0 24
32.0 120
...
```

Once you get this logic right, the rest should be easy.

For efficiency's sake, don't use *power* and *factorial* functions. Instead, compute the required powers and factorials incrementally, as part of the loop's logic. As it turns out, this is not only a matter of efficiency; Java cannot calculate *N!* correctly on integers greater than 12.

2. Collatz conjecture

A *hailstone sequence* is created as follows: Start with some positive integer, let's call it *seed*, and obtain a sequence of numbers by following these rules:

(1) If the current number is even, divide it by 2; otherwise, multiply it by 3 and add 1;
(2) Repeat.

For example, here are the first 8 hailstone sequences (the first number in each sequence is the seed):

1, 4, 2, 1, 4, 2, 1, ...
2, 1, 4, 2, 1, 4, 2, 1, ...
3, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ...
4, 2, 1, 4, 2, 1, ...
5, 16, 8, 4, 2, 1, 4, 2, 1, ...
6, 3, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ...
7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, …

It appears, from these examples, that hailstone sequences tend to reach the number 1. Indeed, the Collatz conjecture (1937) states that for any seed *N*, a hailstone sequence will reach 1.

Write a program that, given *N*, verifies the Collatz conjecture for all seeds between 1 and *N*. Call the program `Collatz`.

The program takes two command-line arguments: the highest seed *N*, and a string which we call *mode.* The string can either be "`v`" (*verbose)* or "`c`" (*concise*). In *verbose* mode, the program prints all the sequences, from *seed* = 1 to *seed* = *N*. For each sequence, the program prints all the values until the sequence reaches 1. Next, the program prints the number of steps it took to reach 1. Finally, the program prints a summary line. In *concise* mode, the program prints only the summary line.

If the program terminates and prints the summary line, it verifies the Collatz conjecture up to *N*. Here are two examples of the program execution:

```
% java Collatz 7 v
1 4 2 1 (4)
2 1 (2)
3 10 5 16 8 4 2 1 (8)
4 2 1 (3)
5 16 8 4 2 1 (6)
6 3 10 5 16 8 4 2 1 (9)
7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 (17)
The first 7 hailstone sequences reached 1.

% java Collatz 1000000 c
The first 1000000 hailstone sequences reached 1.
```

Proposed implementation: Start by writing a loop that prints a hailstone sequence for a given seed value. Test the program for several seed values. Next, nest this loop inside an outer loop that varies the seed from 1 to *N*. Finally, handle the verbose/concise requirement.

Note: if `str` is a variable of type `String`, and you want to check if it equals, say, the string **"x"**, you can use the method call `str.equals("x")`.

3. The birthday problem

What is the probability that, in a group of *N* persons, at least two persons share the same birthday? To answer this question, we can start by computing the probability that none of the *N* persons were born on the same day:

$$\overline{p}(N) = 1 \times (1 - \frac{1}{365}) \times (1 - \frac{2}{365}) \times ... \times (1 - \frac{N-1}{365})$$

It follows that the probability that at least two out of *N* persons share the same birthday is $1 - \bar{p}(N)$.

If you don't understand the probabilistic reasoning, don't worry about it. You will take a probability course later in the program.

Write a program named `Birthday` which contains a function with the following signature

```
Public static double probSameBirthday(int n)
```

The function returns the probability that at least two out of *n* persons share the same birthday. Next, write a `main` function that calculates the minimal *n* such that this probability is at least 0.5.

Note 1: For efficiency's sake, compute 1/365 once, and use this value in the rest of your calculations.

Note 2: In this and in the following programming exercises, if we don't specify or demonstrate how the output of a program should look like, you should use your judgement to design and implement a good-looking and sensible output.

4. Calendar

In this exercise you have to complete the code in the supplied `Calendar.java` file. For more information about each of the functions mentioned below, read the class documentation (in the file).

A. Complete and test the `isLeapYear` function. We wrote this function in class.

B. Complete and test the `nDaysInMonth` function. Proposed implementation: Use either `switch`, or nested `if-else`. This function makes use of the `isLeapYear` function.

C. Write a `main` function that gets a *year* as a command-line argument, and prints how many days are in each month in that year. Required: use a loop to print the months. Here are two separate examples of the program's execution:

```
% java Calendar 2018
Year 2018 is a common year
Month 1 has 31 days
Month 2 has 28 days
Month 3 has 31 days
Month 4 has 30 days
Month 5 has 31 days
Month 6 has 30 days
Month 7 has 31 days
Month 8 has 31 days
Month 9 has 30 days
Month 10 has 31 days
Month 11 has 30 days
Month 12 has 31 days
```

```
% java Calendar 2020
Year 2020 is a leap year
Month 1 has 31 days
Month 2 has 29 days
Month 3 has 31 days
Month 4 has 30 days
Month 5 has 31 days
Month 6 has 30 days
Month 7 has 31 days
Month 8 has 31 days
Month 9 has 30 days
Month 10 has 31 days
Month 11 has 30 days
Month 12 has 31 days
```

5. Random integers

In this exercise you have to write a program named `TestRandom.`

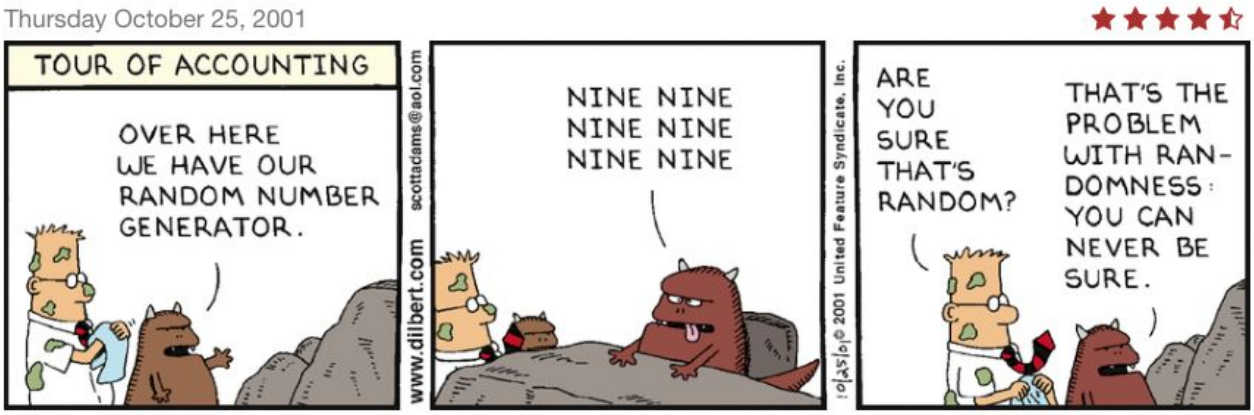
A. Write a function, `random4`, that returns, pseudo-randomly, one of the integers 0, 1, 2, or 3. Proposed implementation: Use Java's `Math.random` function, along with some `if` logic.

B. It's interesting to check how random the returned values really are. Write a `main` function that simulates the act of drawing *N* pseudo-random integers from {0,1,2,3}, and prints how many times each number came

up. If Java's `Math.random` function works well, we should expect to see similar counts. *N* should be a command line argument.

Note 1: the larger *N* is, the more the four counts should be similar to each other. This is a manifestation of the "law of large numbers".

Note 2: We said that “If Java's `Math.random` function works well, we should expect to see similar counts”. The opposite statement is not true. For example, consider a program that returns the sequence 0,1,2,3,0,1,2,3,0,1,2,3,..., for ever. This program passes the equality test with flying colors, but is a very poor random number generator.



(Dilbert Comic, by Scott Adams, Dilbert © 2018, Andrews McMeel Syndication)

Submission

Before submitting your solution, inspect your code and make sure that it is written according to our **Java Coding Style Guidelines**. Also, make sure that each program starts with the program header described in the **Homework Submission Guidelines**. Both documents can be found in the Moodle site, it is your responsibility to read them. Any deviations from these guidelines will result in points penalty. Submit the following five files only:

- `CalcExp.java`
- `Collatz.java`
- `Birthday.java`
- `Calendar.java`
- `TestRandom.java`

Deadline: Submit Homework 2 no later than November 4, 23:55. You are welcome to submit earlier.