Homework 2

1. Divisors

We say that integer *d* divides integer *x* is there is an integer *k* such that $d \cdot k = x$. If so, we say that *d* is a *divisor* (or a *factor*) of *x*. Write a program (`Divisors.java`) that gets a command-line argument (an int) and prints all the divisors of that number. Here are some examples of the program's execution:

```
% java Divisors 18
1
2
3
6
9
18

% java Divisors 239
1
239
```

Tip: Consider using the modulo operator `%`.

2. Reversing a string

(10 points) Write a program (`Reverse.java`) that takes a command-line argument (a string), prints it in reversed order, and then prints the middle character in the given string. Here are two examples of the program's execution:

```
% java Reverse abc
cba
The middle character is b

% java Reverse abcxyz
zyxcba
The middle character is c
```

Tips: Use the string functions `str.length()` and `str.charAt(`i`)`. You can find their API documentation by searching the Internet for "`java 20 string`". Let's assume that the length of the input string is n. The program can be implemented using either a `for` loop, or a `while` loop, that goes backward, from n – 1 to 0. For your education, it's important that you try both implementations.

3. Lucky streak

Write a program (`InOrder.java`) that generates and prints random integers in the range [0,10), as long as they form a non-decreasing sequence. Here are some examples of the program's execution:

```
% java InOrder
3 5

% java InOrder
8

% java InOrder
5 7 7 8
```

Tips: The first generated number always gets printed. Consider using a `do-while` loop for the rest of the program. See the lecture notes how to generate a random integer in the range [*n* , *m*), using `Math.random`.

Play with (execute) this program many times, to get a feeling of the frequency of random lucky streaks (scoring a consecutive number of shots in a basketball game, investing consecutively in good stocks, and so on).

4. Perfect Numbers

A number is said to be *perfect* if it equals the sum of all its divisors, except for the number itself. For example, the divisors of 6 except for 6 are 1, 2, and 3, and 6 = 1 + 2 + 3. Therefore 6 is a perfect number. Write a program (`perfect.java`) that takes an integer command-line argument value, say *N*, and checks if the number is perfect. Here are some examples of the program's execution:

```
% java Perfect 6
6 is a perfect number since 6 = 1 + 2 + 3

% java Perfect 8
8 is not a perfect number
```

Test your program on, at least, the following numbers: 6, 24, 28, 496, 5002, 8128 (four of these numbers are perfect). You can find a list of perfect numbers in the Internet, and use your program to verify that some of them are indeed perfect.

Tips: We suggest the following strategy. When you get a number, say 24, start by building the string "`24 is a perfect number since 24 = 1`". Next, enter a loop that looks for all the divisors of 24 (very similar to what you did in the `Divisors` program). When you find a divisor, append " + " and this divisor to the end of the string. At the end of the loop, check if 24 is indeed a perfect number. If so, print the string that you've constructed all along. If 24 is not a perfect number, ignore the string and print the negative response.

5. Damka Board

Write a program (`DamkaBoard`) that takes an integer command-line argument *n*, and prints an *n*-by-*n* version of a “damka board” (also known as a “checkerboard”). Here are two examples of the program’s execution:

```
% java DamkaBoard 4           % java DamkaBoard 6
* * * *                       * * * * * *
 * * * *                       * * * * * *
* * * *                       * * * * * *
 * * * *                       * * * * * *
                              * * * * * *
                               * * * * * *
```

Tips: Use Java’s `print` function to print each line, incrementally. Use `println` to skip to the next line.

6. One of Each

Some couples have a strong sense of balance: They keep having children until they have at least one boy and at least one girl. Write a program (`OneOfEach.java`) that simulates this behavior. Assume there is an equal probability (0.5) of having either a boy or a girl in each birth. Here are some examples of the program's execution:

```
% java OneOfEach
g g g b
You made it... and you now have 4 children.

% java OneOfEach
b g
You made it... and you now have 2 children.

% java OneOfEach
b b b b b b b b b b b b g
You made it... and you now have 13 children.
```

Tip: Write a loop that (1) Uses the `Math.random` function, and (2) Uses two Boolean variables for recording the fact that a boy, or a girl, where born.

7. One of Each Stats

Start by playing with (executing) the `OneOfEach.java` from the previous exercise about 20 – 30 times. Get a feeling of the statistical results of this family building strategy. Note that each run simulates an experiment in which a different family is formed.

Now write a program (`OneOfEachStats1.java`) that takes an integer command-line argument, say *T*. In each of *T* independent experiments, simulate a couple having children until they have at least one boy and one girl. Use the results of the *T* experiments to compute the *average number of children* that couples who follow this strategy end up having. In addition, compute how many couples had 2 children, 3 children, and 4 or more children. Finally, compute the most common number (also known in statistics as *mode*) of children in a family (if there is a tie, print only the first most common number of children). As before, assume that the probability of having a boy or a girl in each trial is 1/2. Here are some examples of the program's execution (your program will most likely generate other results, because of the randomness):

```
% java OneOfEachStats1 3
Average: 4.333333333333333 children to get at least one of each gender.
Number of families with 2 children: 1
Number of families with 3 children: 0
Number of families with 4 or more children: 2
The most common number of children is 4 or more.

% java OneOfEachStats1 10
Average: 2.7 children to get at least one of each gender.
Number of families with 2 children: 5
Number of families with 3 children: 3
Number of families with 4 or more children: 2
The most common number of children is 2.

% java OneOfEachStats1 1000
Average: 3.045 children to get at least one of each gender.
Number of families with 2 children: 488
Number of families with 3 children: 259
Number of families with 4 or more children: 253
The most common number of children is 2.
```

Tips: Use a `for` loop for running the `T` simulations. In each iteration, execute the same logic as that of the `OneOfEach` program (copy-paste the code of `OneOfEach` into the code of `OneOfEachStats`). Although it's not required, we suggest keeping (at least some of) the print statements of `OneOfEach`, for debugging purposes. When you think that the `OneOfEachStats` program behaves well, you can eliminate, or comment out, these print statements.

Statistical observation: As *T* increases, we expect the average number of children per family to converge to a stable average. Run the program with T = 3, 10, 100, 100000 and 1000000, to watch how the average converges to a stable value. What is this value?

8. One of Each Stats (final version)

The final version (`OneOfEachStats.java`) is almost identical to the previous version (`OneOfEachStats1.java`). The only difference is this: When developing a program that generates random numbers, like a computer game, we must create a version of the program that, when executed, *always generates the same random numbers*. This version enables testing the program in a systematic and predictable way.

In Java, this can be done by using the services of a class named `Random`. Before using this class, you have to `import` it into your program. We'll discuss working with such classes later in the course. For now, simply follow the guidelines that we wrote in the given program skeleton (`OneOfEachStats1.java`).

Submission Instructions for Git Classroom

Preparing Your Submission:

1. Code Formatting: Ensure all your Java code adheres to our Java Coding Style Guidelines (read the document in the Misc section in Moodle). Your repository should include the following Java files:

```
- Divisors.java
- Reverse.java
- InOrder.java
- DamkaBoard.java
- Perfect.java
- OneOfEachStats.java
```

2. Code Documentation PDF:
   - Create a PDF document (`HW2Code.pdf`) containing all your programs' code.
   - Each program should be on a separate page, with proper indentation preserved.
   - Use the font Consolas or Arial, size 12, for code in the PDF to ensure readability.
   - You can use "paste special" options to transfer code from your editor to your word processing software. The final document must have well-indented and easily readable code.

3. Repository Structure:
   - Your Git repository should contain all the Java files and the `HW2Code.pdf`.
   - Ensure your repository is well-organized, with a clear structure and descriptive commit messages.

Submitting Your Work:

1. Accepting the Assignment:
   - Start by accepting the assignment through the link provided by the instructor. This will automatically create a repository in your Git Classroom account.

2. Cloning the Repository:
   - Clone this repository to your local machine to begin working on the assignment.

3. Committing and Pushing Changes:
   - Work on your assignment locally, committing changes to your local repository.
   - Once ready to submit, push these changes back to the Git Classroom repository.

4. Confirm Submission and Viewing Feedback:
   - After pushing to Git Classroom, verify that your files are correctly uploaded and visible in your Git Classroom repository.
   - Your submission should include the Java files (`Divisors.java, Reverse.java, InOrder.java, DamkaBoard.java, Perfect.java, OneOfEachStats.java`) and the `HW2Code.pdf`.
   - View your submission, feedback, and grades directly within Git Classroom.

Submission deadline:

January 04, 2024, 23:55.
Note: Late submissions might not be accepted. Ensure you push your commits well before the deadline.