Homework 2

1. Ordered

(5 points) Write a program (`Ordered.java`) that reads three integer command-line arguments whose values are, say *x*, *y*, and *z*. If the three values are strictly ascending ($x < y < z$) or strictly descending ($x > y > z$), the program prints `true`. Otherwise, the program prints `false`. Here are some examples of the program's execution:

```
% java Ordered 10 17 49
true

% java Ordered 53 40 12
true

% java Ordered 10 49 17
false

% java Ordered 5 19 19
false
```

Implementation tips: (our implementation tips are always *proposals*. You don't *have* to follow them). Define two boolean variables named, say, `ascending` and `descending`, and compute their values. Then use these variables to decide if to `print` true or `false`.

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
```

Implementation tips: Use the string functions `str.length()` and `str.charAt(`*i*`)`. You can read about them by consulting the `String` class API (search the Internet for "`java 16 string`"). The program can be implemented using either a `for` loop that goes backward, or a `while` loop that goes backward. Implement the program using a `for` loop. Then write a second implementation

that uses the `while` loop (`Reverse1.java`). For your education, it's important that you write and test both implementations (`for` and `while`).

3. Damka Board

(10 points) Write a program (`DamkaBoard`) that takes an integer command-line argument *n*, and prints an *n*-by-*n* "damka board" (also known as a "checkerboard"). The board consists of a total of n^2 asterisks (כוכביות), arranged in *n* rows and *n* columns. Each row has 2*n* characters, alternating between asterisks and spaces. Here are two examples of the program's execution:

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

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

Implementation tips: Use Java's `print` function to print each line incrementally. Use `println` to skip to the next line.

4. Divisors

(Basic version of the next program, no points) The *divisors* of a number are all the numbers that divide it, including 1 and excluding the number itself. 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

% java Divisors 239
1
```

5. Perfect Numbers

(30 points) A number is said to be *perfect* if it equals the sum of all its divisors. For example, the divisors of 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. Hint: 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.

Implementation 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. This loop can be identical 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, you will know 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 that you've constructed and print instead `"24 is not a perfect number"`.

6. One of Each

(Basic version of the next program, no points) 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.
```

Implementation tip: The logic of this program is similar to that of the `Perefect` program. Think about how you want to terminate the loop – there are several ways to do it.

Play with (execute) the program about 20-30 times, and get a feeling about the statistical results of this family building strategy. Note that each run simulates how a family a formed.

7. One of Each Stats (version 1)

(Almost final version, no points) 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, 3, 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.
```

Implementation 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.

8. One of Each Stats (final version)

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

In Java, this can be done by using 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

Before submitting your work for grading, make sure that your code is written according to our Java Coding Style Guidelines. Submit the following files only:

- `Ordered.java`
- `Reverse.java`
- `DamkaBoard.java`
- `Perfect.java`
- `OneOfEachStats.java`

Compress the five files into a file named `HW2.zip`, and upload the single zip file using Moodle. **Submission deadline**: October 31, 2021, 23:55.

Get feedback: To get feedback (without grading) about your programs before submitting them, use GETFEED, anytime, as many times as you want.