University of Saskatchewan
Department of Computer Science
176 Thorvaldson Building
110 Science Place, Saskatoon, SK, S7N 5C9, Canada
Telephone: (306) 966-4886, Facimile: (306) 966-4884

Assignment 2

Programming in Java

Date Due: September 26, 2022, 11:59pm **Total Marks: 59**

General Instructions

- **Read these instructions carefully, especially the first assignment. You are responsible for all the instructions here.** They will not change in arbitrary and punitive ways over the semester; any changes will be highlighted.
- **This assignment is individual work.** You may discuss questions and problems with anyone, but the work you hand in for this assignment must be your own work.
- **Assignments are being checked for plagiarism.** We are using state-of-the-art software to compare every pair of student submissions.
- **Make sure your name and student number appear at the top of every document you hand in.** These conventions assist the markers in their work. Failure to follow these conventions will result in needless effort by the markers, and a deduction of grades for you.
- **Assignments must be submitted to Canvas.** It is your responsibility to ensure that your upload worked correctly. See *Instructions for Submission* for more details.
- **The assignment is due at the time stated above.** The due date should appear on your Canvas calendar on the day it is due.
- **Canvas has a built-in late penalty mechanism, which will automatically apply to any files submitted after due date.** After the grace period, 0.6% of your grade will be deducted per hour with 15% deductions per day, increasing by 15% every day thereafter.
- **You do not need to obtain permission to submit an assignment after the due date.** Canvas will be configured to accept submissions for a week after the due date. Extensions for medical and compassionate reasons can still be granted; contact your instructor as usual.
- **Programming questions must be written in Java.** Do not submit IntelliJ projects (unless the assignment says so directly). Do not submit the compiled `.class` files (unless the assignment says so directly). If you submit an IntelliJ project when it was not required, you will get zero for that question!
- **In general, non-programming questions must be submitted as either `.txt` or `.pdf` files.** If other file types are used, you may receive a grade of zero for that question. If the marker cannot open a file because you did not follow these instructions, you will definitely get a zero for anything in that file.

Instructions for submissions

- **Assignment Submissions**: You will submit a single ZIP file containing all your questions solution files. This will help mitigate the problem of Canvas renaming your files if you resubmit something. Please see *What to Hand In* for each question.
  - Name the file `A2-abc123-CMPT270.zip`, where you should use your own NSID instead of `abc123`. This naming convention will assist the markers with their work, and we encourage all students to adopt it.
  - Each question will describe one or more files to submit. These are to be included in the same ZIP file described above. All the files mentioned by all questions in a single assignment go into one ZIP file that you submit.
  - Each question will mention the name of the file(s) to submit, and file format requirements. Failure to follow file format requirements may result in severe mark deductions. If you change or modify any file you've previously submitted, you have to recreate the whole ZIP file, and resubmit it.
  - You must submit a ZIP file, and no other archival format (e.g., no RAR, 7ZIP, etc.). If you use these other tools, and rename it with ZIP, the markers will not be able to open it, and you will get zero, because it could not be opened.

Version History

- **20/09/2022**: released to students

Question 1 (15 points):

Purpose: To build upon a simple interpreter and test it. To practice Java's programming constructs. To see what a virtual machine is like, even if remotely.

Degree of Difficulty: Easy. There is a lot to read, but it is not difficult to implement. There are plenty of opportunities for you to make mistakes that you have to correct, and that's part of the work!

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 270. Solutions will be made available to students registered in CMPT 270 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 270 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

A simple calculating language: Va

The end goal of this question is a Java program that interprets a very simple virtual machine language called Va. Don't worry, this is not a difficult task. It is sufficiently interesting to make learning Java syntax somewhat fun. You can do this with the Java tools we've learned in Weeks 01-03. You will not need objects or classes, or any advanced data structures. The interpreter will work line-by-line: the program will prompt the user for each line, and will execute each line. Details about the language are below, but it's basically a very simple assembly language. It's a bit unusual in that it's interpreted, but we haven't covered File Input/Output yet, so we won't be opening and reading files for this question. That would be ideal, but we won't use File I/O in this assignment.

Va Memory Model The Va virtual machine has the following memory model:

- A single register, which can store a `double` value. Most of the Va language commands affect the register. (see below)
- A heap that can store exactly 5 `double` values. Values can be moved into and out of the heap through Va language commands. We could make the heap as large as we like, but for this assignment we will keep it small. The heap can be indexed using integer addresses, starting at zero.

Va Commands The commands of the Va language are not like Java statements, which has assignment statements and expressions, and loops and functions, etc. In Va, we only have very simple commands. Each command is an instruction in assembly language, for a very simple kind of virtual machine. To describe the commands, we have to give the syntax of the commands, and then clearly describe the behaviour or effect of the commands. In the description that follows, the command keywords will be written in upper case, e.g., SET. Commands sometimes take an argument, which depends on that the script is trying to do. For arguments, we will use italics, e.g., v. In this situation, the v represents something that would be typed by a user. There are only two types of arguments we will use: integer literals (int), and floating point literals (double). Integers will be used only for indexing into the heap (see below), and floating point values for calculations.

We'll start with the SET and TELL commands, which Va uses to interact with the user.

Command	Effect
`SET` v	The given value v is stored as the value of the register.
`TELL`	Display the value of the register on the console

The Va language will also have 4 normal arithmetic commands that affect the register. These commands take a single argument, v. In the table below, the syntax is given on the left, and the effect is stated on the right.

Command	Effect
`ADD` v	$R = R + v$
`SUB` v	$R = R - v$
`MUL` v	$R = R \times v$
`DIV` v	$R = R/v$ (floating point division)

In this table, the v represents a literal numeric floating point value, like 3.1415, typed in by the user. Also, the symbol R represents the register of the machine. For example, the command `ADD 4`, adds the value 4 to whatever number is already in the register. Notice that the commands mention the value v, but not R. That's because all of the arithmetic commands affect R, so we don't need to specify that in the command itself; the R is implied.

There are only two commands that will affect the heap, which is a block of memory capable of storing 5 double values.

Command	Effect
`STORE` i	$\text{heap}[i] = R$
`LOAD` i	$R = \text{heap}[i]$

Note that the symbol i here represents an integer value typed by the user. In words, the STORE command copies the value currently in the register R to one of the locations in the heap, as given by an integer index. Likewise, the LOAD command copies the value from the heap at location i to the register R.

Finally, there are 4 indirect arithmetic commands that affect the register using values taken from the heap. Here, the commands take a single integer argument.

Command	Effect
`ADDI` i	$R = R + \text{heap}[i]$
`SUBI` i	$R = R - \text{heap}[i]$
`MULI` i	$R = R \times \text{heap}[i]$
`DIVI` i	$R = R/\text{heap}[i]$ (floating point division)

These four commands have the suffix "`I`", meaning indirect. They are indirect because the argument i is an index integer, and refers to a location on the heap. For example, the command `ADDI 4`, adds the value currently stored in the heap at index 4 to whatever number is already in the register. These commands still affect the register only; the heap values are used but not changed by these commands.

The Va virtual machine has a simple debugger, the command STATE. This command takes no arguments, and displays the content of the virtual machine: the register, and all the values on the heap. See the examples below for the demonstration of what this command should do. The STATE command shows us what the register and the heap values are, and its primary usefulness would be in debugging a Va program.

Finally, the user can terminate the Va machine using the command HALT.

Example 1 Now that we have seen the components of the Va language, let's see some examples. In the examples that follow, notice that the Va interpreter uses the prompt >> to indicate that the user can type something.

Here's a simple demo that calculates $5!$:

```
>> SET 1
>> MUL 2
>> MUL 3
>> MUL 4
>> MUL 5
>> TELL
```

```
120.0
>> HALT
```

The interpreter is displaying the prompt >> and the user types the commands and ends the line by using the return key. In this example, the first line stores the floating point value 1 in the register. The following 4 lines do the work of multiplication, using literal factors 2,3,4,5, and the TELL command asks the interpreter to display the value in the register. As we can see, the value is equal to $5!$.

Example 2 The previous example did not use the heap at all. Here's an example using the heap, also showing the STATE command:

```
>> SET 100
>> STORE 0
>> SET -100
>> STORE 1
>> STATE
State:
  register = -100.0
  heap[0] = 100.0
  heap[1] = -100.0
  heap[2] = 0.0
  heap[3] = 0.0
  heap[4] = 0.0
>> HALT
```

As you can see, the STATE command displays the register and all the values in the heap. Note that the STORE command copies the value currently in the register to a given location in the heap.

Example 3 The following example extends the previous one.

```
>> SET 100
>> STORE 0
>> SET -100
>> STORE 1
>> STATE
State:
  register = -100.0
  heap[0] = 100.0
  heap[1] = -100.0
  heap[2] = 0.0
  heap[3] = 0.0
  heap[4] = 0.0
>> LOAD 0
>> STORE 4
>> LOAD 1
>> STORE 0
>> LOAD 4
>> STORE 1
>> STATE
State:
  register = 100.0
  heap[0] = -100.0
  heap[1] = 100.0
  heap[2] = 0.0
  heap[3] = 0.0
  heap[4] = 100.0
>> HALT
```

This example stores the values 100 and -100 in the heap at locations 0, and 1, exactly as the previous example did. This is visible by the first use of the STATE command. After the first STATE command, the values in locations 0 and 1 are swapped, using location 4 as temporary storage. The result can be verified by viewing the output presented by the second STATE command.

Example 4 Consider the following simple Java code:

```
double x = 1.0;
double prod = 1.0;
x = x + 1;
prod = prod * x;
```

There are two variables, and two initializations, followed by a couple of assignment statements.

We can express this same calculation using the Va language. We will use location 0 for `x`, and location 1 for `prod`, as follows:

```
SET 1
STORE 0
STORE 1
ADD 1
STORE 0
MULI 1
STORE 1
```

In the above sequence:

- Lines 1-3 store the value 1 in the location 0 for `x` and location 1 for `prod`
- Lines 4-5 add 1 to to the location 0 for `x`
- Lines 6-7 updates the location for location 1 for `prod` by multiplying by location 0 `x`.

Notice how our assembly language uses indices instead of variable names!

Example 5 If we had loops in our language (we don't!), we might wrap a loop around lines 4-7 from the previous example to accomplish the factorial calculation. But since we don't have loops, we can simply copy/paste lines 4-7 a few times to calculate $5!$, as follows:

```
1  SET 1
2  STORE 0
3  STORE 1
4  ADD 1
5  STORE 0
6  MULI 1
7  STORE 1
8  LOAD 0
9  ADD 1
10 STORE 0
11 MULI 1
12 STORE 1
13 LOAD 0
14 ADD 1
15 STORE 0
16 MULI 1
17 STORE 1
18 LOAD 0
19 ADD 1
20 STORE 0
21 MULI 1
22 STORE 1
23 LOAD 1
24 TELL
```

You can see that the Lines 4-8 are repeated 4 times, increasing `x` (location 0) from 1 to 5, then multiplying with the value in `prod` (location 1) each time. At the end, the displayed value is 120.

Adding loops to Va would make it a more powerful language, but that would be a bit more challenging than we need. Our purpose is to work on an interesting program to practice our Java coding skills.

Task

Your task is to finish implementing the Va commands shown above from the starter code given. Commands SET, TELL, STORE, ADD, SUB, and HALT, the memory model, as well as the user input loop have already been implemented. Your task is to implement the missing commands LOAD, MUL, DIV, ADDI, SUBI, DIVI, MULI and STATE.

Hints:

- Some commands have already been implemented, so additional commands can easily be added by just adding additional `else if` statements
- The starter code ignores user input errors, so you do not need to worry about improper commands.
- If your program crashes because of Java code you wrote, you have to fix it.

Demonstration

When you are done, you should be able to use your interpreter to do very simple calculations. To demonstrate that your program works, show it working by calculating the following quantities:

- The quantity:
$$\frac{n \times (n+1)}{2}$$
for $n = 12$.
  - First, do the trivial thing and calculate $(12 \times 13)/2$.
  - After, use the heap, as shown in the examples above. Store the value 12 in the heap at location 0, and then come up with a sequence of commands that treats location 0 as if it were a variable. Your sequence should work no matter what value is stored at location 0. I.e., you've written a simple program.
- The quantity:
$$5^2 + 6^2 + ... + 9^2$$
.
  - First, do the trivial thing and calculate $25+36+49+64+81$. This is similar to the simple calculation for factorial in the example above.
  - After, use the heap, as shown in the examples. Use locations in the heap as "variables." You could use a variable for a value that is increasing from 5 to 9, and a variable to store the square of each number, and finally, a variable to store the sum of the squares as they are calculated.

What to Hand In

- Your implementation of the Va interpreter: `a2q1.java`.
- The demonstration of your program working, as described above. Copy/paste from the console to a text file and hand in as `a2q1_demo.txt`

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- (8 marks) Your Java program implements the missing commands of the Va language (LOAD, MUL, DIV, ADDI, SUBI, MULI, DIVI, STATE)
- (2 marks) Documentation. Inline comments are judiciously used, neither too much nor too little.
- (5 marks) Demonstration.
  - You included a text file that shows the calculations mentioned in the Demonstration section above.
- We have not covered testing yet, so there is not requirement to submit testing. You should of course try to test as much as you can.

Question 2 (19 points):

Purpose: To further build on a simple interpreter and test it.

Degree of Difficulty: Easy. Again, this is not hard, but it will take time!

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 270. Solutions will be made available to students registered in CMPT 270 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 270 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

Do not start this question until you have completed question 1.

The objective of this question is to add some additional functionality to the Va language. We will implement five commands that can perform comparison operations, and will store a `boolean` value in memory.

BEFORE STARTING THIS QUESTION, copy the code from your completed `a2q1.java` into a new file `a2q2.java` to work on. This question builds upon the work done in question 1.

We will add the following commands:

Command	Effect
`LESS` v	$bR = R < v$
`LESSEQ` v	$bR = R <= v$
`EQUAL` v	$bR = R == v$
`GREATEQ` v	$bR = R >= v$
`GREAT` v	$bR = R > v$

In this table, the v represents a literal numeric floating point value, like 3.1415, typed in by the user. The symbol R represents the register of the machine that stores a `double` value, and the symbol bR represents the `boolean` register of the machine. For example, the command `LESS 4`, compares the value of the register v to the literal value 4, and stores the result of this comparison (`true/false`) to the register bR.

The commands above will store the result of the comparison to a new `boolean` register bR. This new register will store the value until it is replaced, meaning that the value stored there will be the result of the last comparison operation done.

Example 1 Now that we have seen the new commands that will be added to the Va language, let's see some examples of them in use. In the examples that follow, notice that the Va interpreter uses the prompt >> to indicate that the user can type something.

Here's a simple demo that performs some simple arithmetic to get the value 15 in the register, then compares that to the literal value 20:

```
>> SET 5
>> MUL 5
>> SUB 10
>> TELL
15.0
false
>> LESS 20
>> TELL
15.0
true
>>
```

In this example, the first two lines performs the arithmetic 5 * 5 and stores the value 25 into the `double` register. The next line subtracts 10 from the value of 25 to store the value 15 in the `double` register. The TELL command now shows the status of the `double` register as 15, and the `boolean` register as false. The LESS command performs the comparison 15 < 20, and stores the result of that comparison (`true`) into the `boolean` register. The last command, TELL, shows that the `boolean` register has been changed to `true` after the comparison.

Example 2 Here's a simple demo that works on the heap, and performs a comparison:

```
>> TELL
0.0
false
>> SET 10
>> STORE 0
>> ADDI 0
>> STORE 1
>> LOAD 0
>> MULI 1
>> EQUAL 200
>> TELL
200.0
true
>> STATE
State:
  register = 200.0
  bRegister = true
  heap[0] = 10.0
  heap[1] = 20.0
  heap[2] = 0.0
  heap[3] = 0.0
  heap[4] = 0.0
>>
```

In this example, we first store the value 10 to heap at index 0. Then, add register value 10 to heap value 10 to get register value 20, which is then stored in heap index 1. We load value 10 in from heap index 0, and multiply register value 10 with heap value 20 to get the value 200 in the register. We then perform the comparison `EQUAL 200`, to check if the register value is equal to the value 200. We then can see that the value in the `boolean` register has been changed to `true`, and heap values have been modified as well.

Task

Your task is to add a new register to the Va program, and implement the five commands shown above. You will also make modifications to the TELL and STATE commands to show the status of the new boolean register.

Hints:

- First you should add the new boolean register. For our purpose it is alright to initialize it to `false`.
- Next, you should modify the TELL and STATE commands to show the status of the new register.
- Now you can work to implement the commands.

NOTE: Make sure that you have the correct order for the comparisons! Consult the table above to make sure you have the register and input values in the correct place.

Demonstration

When you are done, you should be able to use your interpreter to perform comparisons. To demonstrate that your new commands work, show it working by performing the following commands in the order that they appear below:

1. Input the STATE command to show the initialized state of the program.
2. Store the value 120 to the `double` register, followed by inputting the TELL command.
3. Input the command LESS 150, followed by the TELL command.
4. Input the command LESSEQ 99, followed by the TELL command.
5. Input the command GREAT 99, followed by the TELL command.
6. Input the command GREATEQ 125, followed by the TELL command.
7. Input the command EQUAL 120, followed by the TELL command.

What to Hand In

- Your implementation of the Va interpreter with the additional and modified commands as described above: `a2q2.java`.
- The demonstration of your new commands working, as described above. Copy/paste from the console to a text file and hand in as `a2q2_demo.txt`

Be sure to include your name, NSID, student number and course number at the top of all documents.

Evaluation

- (10 marks) Your Java program implements the new commands (LESS, LESSEQ, EQUAL, GREAT, GREATEQ) and the new boolean register.
- (2 marks) Your Java program modifies the TELL and STATE commands to show the status of the new register.
- (2 marks) Documentation. Inline comments are judiciously used, neither too much nor too little.
- (5 marks) Demonstration.
  - You included a text file that shows the calculations mentioned in the Demonstration section above.
- We have not covered testing yet, so there is not a requirement to submit testing. You should of course try to test as much as you can.

Question 3 (25 points):

Purpose: To build a program with multiple methods, practising basic Java programming skills, especially arrays!

Degree of Difficulty: Moderate. Don't leave this to the last minute. This task involves an unfamiliar problem, but the program itself is not more difficult than anything you've done in first year.

A Number Square

The following description considers the manipulation of a 3×3 array of integers, which can be considered a kind of simple game. The description will start by describing the rules of the game, and then you'll be told what your task will be.

Consider an 3×3 array of integers, with 3 of each integer 1, 2, and 3, arranged as follows:

1	1	1
2	2	2
3	3	3

This is a kind of a puzzle, where we manipulate the numbers in the square by 'rotating' sub-squares. We will call these actions *sub-square rotations*, for reasons that will be clarified immediately.

A *sub-square* is a group of four numbers that make a smaller square within the larger number square. In our 3×3 number square there are **four** sub-squares. The bold numbers shown below forms the sub-square 0 (top left).

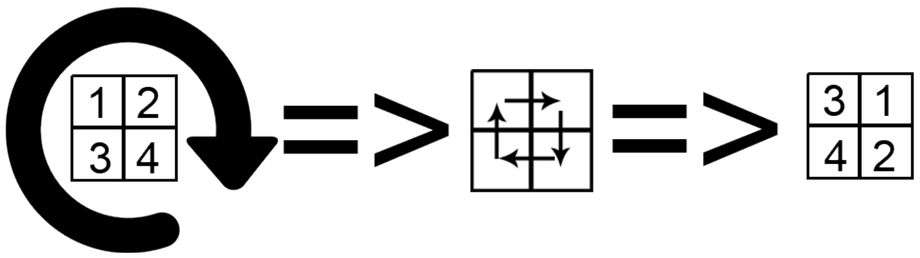
1	**1**	2
3	**2**	2
3	1	3

The four sub-squares are indexed as follows:

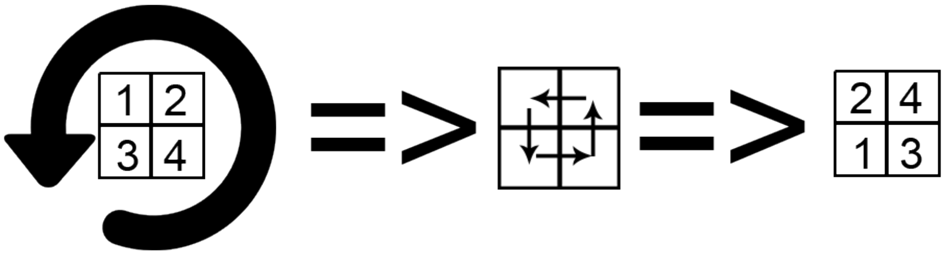
- Sub-square 0 is the top left sub-square.
- Sub-square 1 is the top right sub-square.
- Sub-square 2 is the bottom left sub-square.
- Sub-square 3 is the bottom right sub-square.

In this game, we can rotate sub-squares either right (clockwise), or left (counter-clockwise).

Below shows how a clockwise rotation occurs on a subsquare with four unique values:



And how a counter-clockwise rotation is performed:



These subsquare rotations happen without changing the rest of the square.

1	**1**	2
3	**2**	2
3	1	3

For example, we can rotate sub-square 0 (top left) of the above example square clockwise (right) to get the following:

3	**1**	2
2	**1**	2
3	1	3

Notice that the values in the sub-square have shifted so that the value 3 moved up, the value 2 moved to the left, and the two 1 values are now vertical and on the right of the sub-square. These shifts in position is similar to what would happen if we could rotate the sub-square in place to the right (clockwise).

Likewise, we can rotate a sub-square to the left (counter-clockwise). Below we will rotate sub-square 1 (top right), left (counter-clockwise).

Before rotation:

3	**1**	**2**
2	**1**	**2**
3	1	3

After rotation:

3	**2**	**2**
2	**1**	**1**
3	1	3

We will represent each possible rotation with a code, which we will motivate by example, and then define carefully after the example. A sub-square rotation of the top left sub-square to the right will be represented by the string `'R 0'`. The first letter `'R'` indicates that a sub-square is rotated to the right (clockwise). The digit `'0'` indicates that the first sub-square is rotated (top left).

In general, a rotation will consist of a single letter followed by a single digit.

- `'R i'` means rotate sub-square i to the right (clockwise) one position.
- `'L i'` means rotate sub-square i to the left (counter-clockwise) one position.

Here, i has to be a valid sub-square index, that is: 0, 1, 2, or 3.

Now that we have a code for the rotations, we can implement a kind of puzzle game.

The player is given an array A_1, for example:

```
1 2 1
1 3 2
3 3 2
```

We can use this array as a starting position. The goal of the puzzle is to return the square to the goal position, which is as follows:

```
1 1 1
2 2 2
3 3 3
```

The player can only manipulate the square by choosing and typing a rotation.

For example, here is a transcript from a simple game:

```
Number Square Rotation Game

Initial position:
1 2 1
1 3 2
3 3 2

Enter move: L 3

1 2 1
1 2 2
3 3 3

Enter move: R 0

1 1 1
2 2 2
3 3 3

You win!
```

In this example, the program is displaying everything except for the rotations $L\ 3$ and $R\ 0$, which are typed by the user (the text `Enter move:` is a prompt displayed by the program).

Task

Implement a Java program that allows the user to play this puzzle game. You should start with an initial configuration, and prompt the user for a single rotation. If the user's command is valid, the rotation should be applied. Your program should check to see if the square has been put into the goal position after every rotation.

Hints:

- Define a method to display a 3×3 square on the console.
- Define a method to compare two 3×3 squares, to see if they have all the same values in the same locations. This will be useful for the game, and for your debugging.

- Define a method to perform a right sub-square rotation, and define another method to perform a left sub-square rotation. You could leave them combined, however separating them will make the code more readable.
- Since arrays are pass-by-reference, it need not return anything.
- With these methods written and tested, the game is pretty simple. Start with any arrangement of integers 1, 2, and 3, where there are three each. Then repeat these steps:
  1. Display the square (using the first method).
  2. Prompt for a rotation (using console I/O).
  3. If it's valid, apply the rotation (using one of the other methods).
  4. If the square is in the goal configuration, end the game with a cheerful text!

  Note: Your program doesn't have to decide the rotations. The user decides the rotations.

What to Hand In

- Your implementation of the program: `a2q3.java`.
- A demonstration of a user playing your game, and your program applying rotations. Copy/paste console output to a text file named `a2q3.txt`.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- (15 marks) Your Java program implements the number square rotation game:
  - 3 marks: The main game loop ends with the goal configuration.
  - 3 marks: Your method to display squares is correct.
  - 3 marks: Your method to compare 2 squares is correct.
  - 3 marks: Your method to apply left sub-square rotations is correct.
  - 3 marks: Your method to apply right sub-square rotations is correct.
- (5 marks) Documentation. Methods are documented with Javadoc comments. Inline comments are judiciously used, neither too much nor too little.
- (5 marks) Demonstration.
  - You included a text file that shows the user playing the game by making sub-square rotations. You do not have to show a winning game! Just show the rotations are working.
- We have not covered testing yet, so there is not requirement to submit testing. You should of course try to test as much as you can.

Examples

To make your game more fun, here are some starting squares for the player to work on. These are provided for your amusement, and there are no marks associated with their use.

3	3	2
2	1	1
3	1	2

3	2	3
1	2	1
2	3	1

2	1	3
1	1	2
3	2	3

1	2	3
1	2	3
1	2	3

3	3	3
1	1	1
2	2	2

1	3	3
1	2	3
2	1	2