ITSC 1213 - Magic Square

Introduction

One interesting application of two-dimensional arrays is magic squares. A magic square is a square matrix in which the sum of every row, every column, and both diagonals is the same.

Magic squares have been studied for many years, and there are some particularly famous magic squares. In this lab you will write code to determine whether a square is magic.

A magic square of order n is an arrangement of n^2 numbers in a square with the following properties:

1. The n numbers in all rows, all columns, and on both diagonals sum to the same constant.
2. Each number should only occur once in the magic square.

Here is an example:

2 7 6
9 5 1
4 3 8

In the square above, each number appears only once, and each row, column, and diagonal sums to 15.

Rows:
2 + 7 + 6 = 15
9 + 5 + 1 = 15
4 + 3 + 8 = 15
Columns:
2 + 9 + 4 = 15
7 + 5 + 3 = 15
6 + 1 + 8 = 15
Diagonals:
2 + 5 + 8 = 15
4 + 5 + 6 = 15

The number of integers on each line must be equal to the total number of lines in the square. In other words, the following formats would NOT represent a magic square:

2 7 6 9 5 1 4 3 8

2 7 6 9
5 1
4 3 8

2 7 6 9
5 1 4 3
8

Concepts covered in this lab:

- Traversing 2D arrays
- Testing code

Required files or links

- Square.java - Java source code
- Main method code snippet for testing

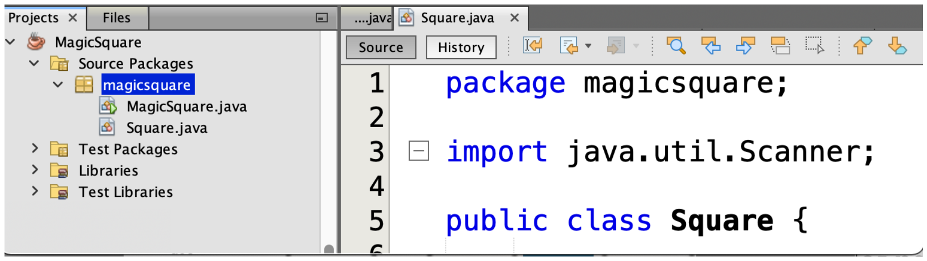
Part A: Setup project and files

1. Create a new project in NetBeans called MagicSquare with a main class. Remember that the main class is the class we will use to add code that will be executed when we run the project. We will refer to this class as the main class but you are free to call it any valid Java class name (e.g., `Main` or `MagicSqaure`).

2. Right-click on the package that has the main class to create a new class. Select New → Empty Java File. Call your new class `Square`, select a package to (create one if you don't have one created already) and click Finish.

3. Copy and paste the code from this code snippet into this file - https://gist.github.com/nanajjar/d8b40b0d287e4dae61ec6be96e4f41ae

   Make sure you include the correct package declaration at the beginning of the file to match what you have in your project



```java
package magicsquare;

import java.util.Scanner;

public class Square {
```

Examine the code you just copied to get a sense for what it is doing. This file contains the shell for a class that represents a square matrix. It contains constructors for initializing a square matrix as a 2D array and methods to read values into the square, print the square, find the sum of a given row, find the sum of a given column, find the sum of the main (or other) diagonal, and determine whether the square is magic. The `read` and `print` methods are given for you; we will work on completing the others in this lab.

- `sumRow(...)`
- `sumColumn(...)`
- `sumMainDiag( )`
- `sumOtherDiag( )`
- `isMagic( )`

A few things to note here:

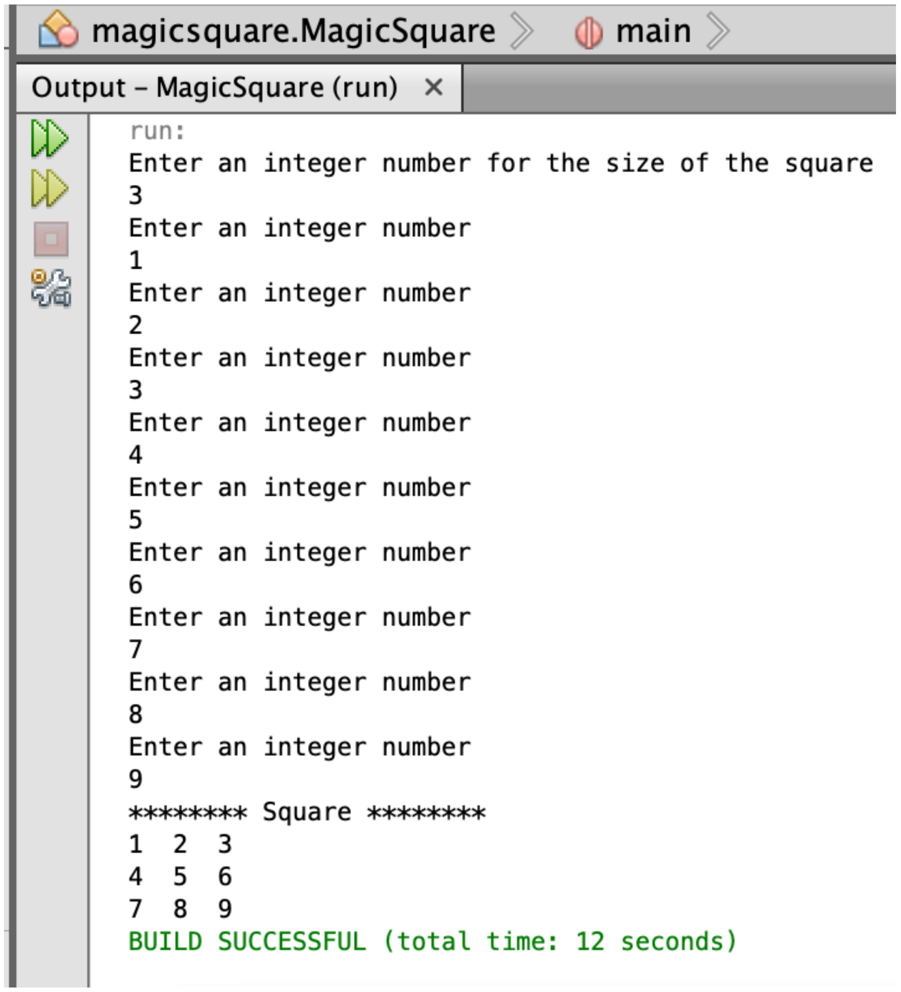
- for this lab we are only concerned in checking the first property of a magic square matrix and not the second one (more about this in part D).
- the `read` method is responsible for setting the values in the matrix/square using user input. It takes a Scanner object as a parameter
- the `print` method is responsible for nicely formatting the printing of the square.

4. Now, switch over to your main class and copy and paste the code from this code snippet - https://gist.github.com/nanajjar/b0af31d6ad8903a872ec81dcc8bf563e into the `main()` method. Read through the code to get a sense for what it is doing. Note that you will need to add the import statement for the `Scanner` class as we are using it in the code snippet you just copied.

5. To make sure you've set things correctly, let's test creating a square using our program. Run the main class, you should be prompted to enter a size for the square matrix to be created, enter a small value to get started (e.g., 3) since you will need to enter a value of each cell. Looking into our main method we can see that we are calling the `readSquare` method which will prompt you to enter

values to fill in the square matrix. At this point it does not matter what values you use. Keep adding numbers as the program will prompt you as many times as needed to fill the matrix.

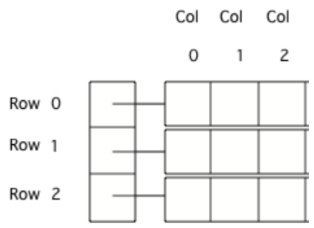
Your output should look something like this:



```
run:
Enter an integer number for the size of the square
3
Enter an integer number
1
Enter an integer number
2
Enter an integer number
3
Enter an integer number
4
Enter an integer number
5
Enter an integer number
6
Enter an integer number
7
Enter an integer number
8
Enter an integer number
9
******** Square ********
1  2  3
4  5  6
7  8  9
BUILD SUCCESSFUL (total time: 12 seconds)
```

6. When your program works and you are satisfied with the result, show your work to the instructional team to get checked off and proceed to Part B.

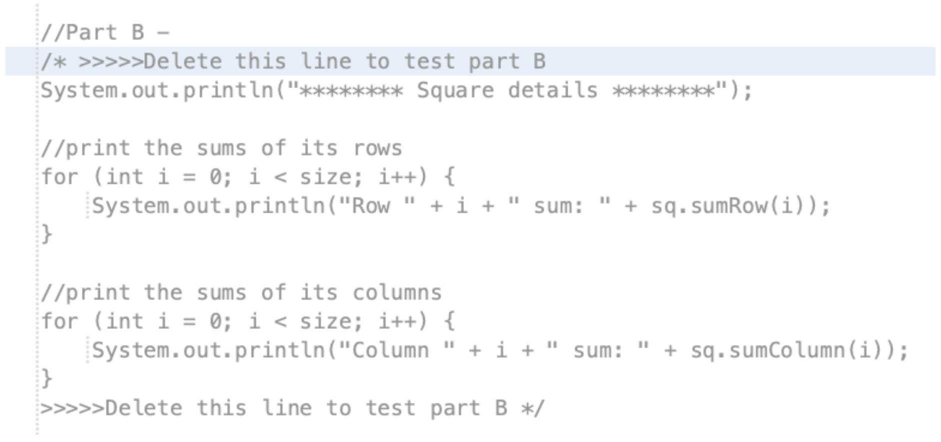
Part B - Sum rows and columns



1. In this part we will work on completing the `sumRow` and `sumColumn` methods of the `Square` class. Each of those methods take an integer parameter that will hold the value for the row or column we want to sum.
2. Start with the `sumRow` method, this method should return the sum of the values in the given row of the square matrix. Remember that in a 2D array if we want to reference all the elements in a row we need to be moving along the columns while keeping the row reference constant. Add logic that will total the columns values for the given row and return it.

```java
//--------------------------------------
//return the sum of the values in the given row
//--------------------------------------
public int sumRow(int row) {
    int total = 0;
    //Add code here!

    return total;
}
```

3. Before we move to the `sumColumn` method, let's try the `sumRow` method. If we verify that this method works it will make the next step go a little faster/smoother since both methods are very similar. Switch over to the main method and delete the two lines that include " >>>>>Delete this line to test part B".



```java
//Part B -
/* >>>>>Delete this line to test part B
System.out.println("******** Square details ********");

//print the sums of its rows
for (int i = 0; i < size; i++) {
    System.out.println("Row " + i + " sum: " + sq.sumRow(i));
}

//print the sums of its columns
for (int i = 0; i < size; i++) {
    System.out.println("Column " + i + " sum: " + sq.sumColumn(i));
}
>>>>>Delete this line to test part B */
```

Run your program and verify that the `sumRow` method does what you expect it to do. Make sure to try different values to ensure the method behaves as expected. Keep in mind that since we didn't fully implement the `sumColumn` method the output for the column sums will display zeros.

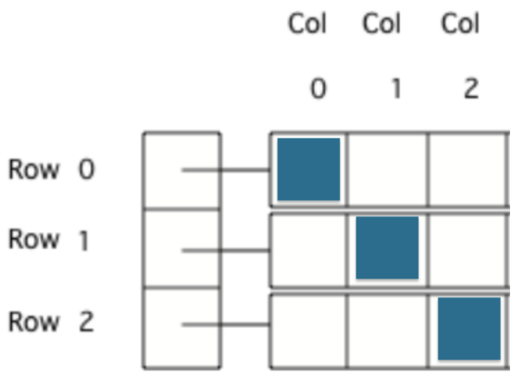
4. When you are satisfied with the `sumRow` method you are ready to implement the `sumColumn` method in the `Square` class. This method should return the sum of the values in the given column of the square matrix. Remember that in a 2D array if we want to reference all the elements in a column we need to be moving along the rows while keeping the column reference constant. Add logic that will total the rows values for the given column and return it.

```java
//--------------------------------------
//return the sum of the values in the given column
//--------------------------------------
public int sumColumn(int col) {
    int total = 0;
    //Add code here!

    return total;
}
```

5. Test your method by running the main program. The output should now include column totals rather than zeros. Run your program a few times using different square sizes and values.
6. When your program works and you are satisfied with the result, show your work to the instructional team to get checked off and proceed to Part C.

Part C - Sum diagonals

1. The last thing we need to do before we are able to determine if a square is a magic square is to sum the two diagonals. In Part B we were traversing the 2D array but only moving either horizontally or vertically (i.e., either across columns or across rows). To get the sum of a diagonal requires us to traverse the 2D array changing both the row and column references.
   Let's take a 3x3 matrix and think of the indices that reference the diagonal elements. Do you see a pattern here?



2. If you determined that we need to only add those elements of the matrix where the row number is equal to the column number then you were correct. Add logic to the `sumMainDiag` method that will accomplish this - total the values for the main diagonal of the matrix and return it.

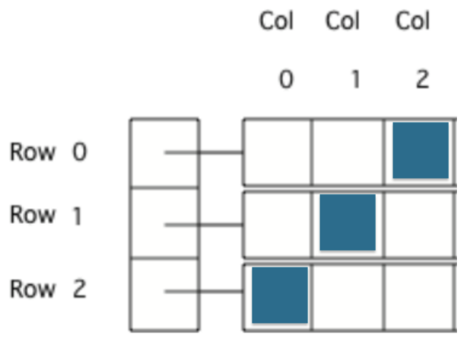
```
//----------------------------------------
//return the sum of the values in the main diagonal
//----------------------------------------
public int sumMainDiag() {
    int total = 0;
    //Add code here!

    return total;
}
```

3. Similar to what we did in Part B we can test this method before completing the `sumOtherDiag` method. Switch over to the main method and delete the two lines that include " >>>>>Delete this line to test part C".

```
//Part C -
/* >>>>>Delete this line to test part C
//print the sum of the main diagonal
System.out.println("Main diagonal sum: " + sq.sumMainDiag());

//print the sum of the other diagonal
System.out.println("Other diagonal sum: " + sq.sumOtherDiag());

>>>>>Delete this line to test part C */
```

4. Run your program and verify that the `sumMainDiag` method does what you expect it to do. Make sure to try different values to ensure the method behaves as expected. Keep in mind that since we didn't fully implement the `SumOtherDiag` method the output from that method will display zero.

5. When you are satisfied with the `sumMainDiag` method you are ready to implement the `sumOtherDiag` method in the `Square` class. This method should return the sum of the values in the secondary diagonal of the square.



6. Again, try to identify the relationship between the row index and the column index that references the secondary diagonal. Remember that it should work regardless of the size of the matrix/square.
   When trying to identify a pattern you might find it helpful to write down a few examples. So, if we take as an example the 3x3 matrix in the figure above and list the index numbers for the diagonal we want to sum they would be:
   Row index 0, column index 2
   Row index 1, column index 1
   Row index 2, column index 0

   We can see here that the two indices are moving opposite each other. One starts from the beginning and goes up until the end of the array, while the other starts from the end of the array and goes down until the beginning of the array.

7. Add logic that will total the rows values for the given column and return it.
   Note: While the logic can utilize two different variables to reference the array indices try to think of a way to represent this pattern using one variable.

```java
//-----------------------------------------
//return the sum of the values in the other ("reverse") diagonal
//-----------------------------------------
public int sumOtherDiag() {
    int total = 0;

    return total;

}
```

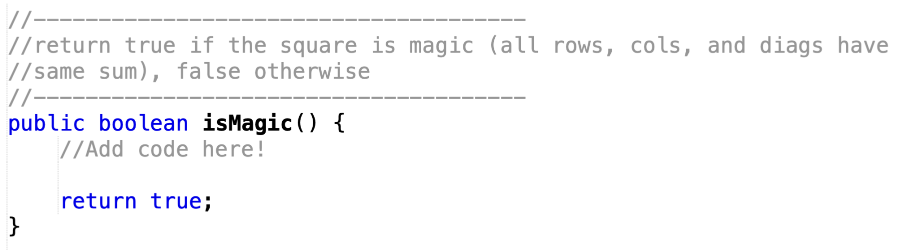
8. Run your program to make sure that your method is working as expected. Fix any bugs you find. Test your code with different values each time to make sure it works regardless of the square size and content.

9. When your program works and you are satisfied with the result, show your work to the instructional team to get checked off and proceed to Part D.

Part D - Is the square a magic one?

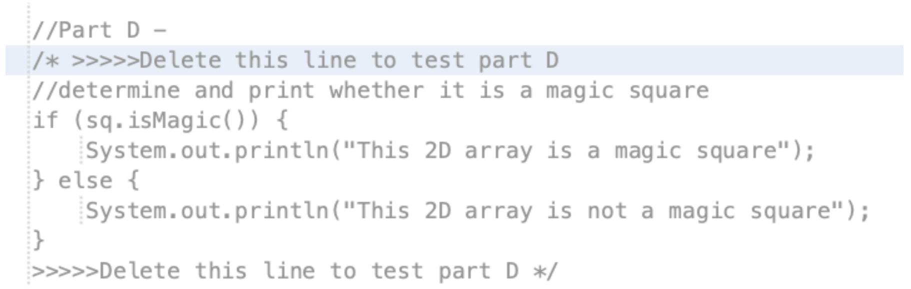
Now that we have all the methods we need to perform the necessary calculations to help us determine whether the square satisfies the first condition for it to be a magic square we can complete the isMagic method. For now since we do not have a method to check if the values of the matrix are all unique we will base this only by checking for the sum of the rows, columns and diagonals.

1. The isMagic method returns true if the sum of every row, every column, and both diagonals is the same and false otherwise. Add logic to the `isMagic` method that will accomplish this. Make sure you are calling the sum methods we implemented in parts B and C to get the sum values.



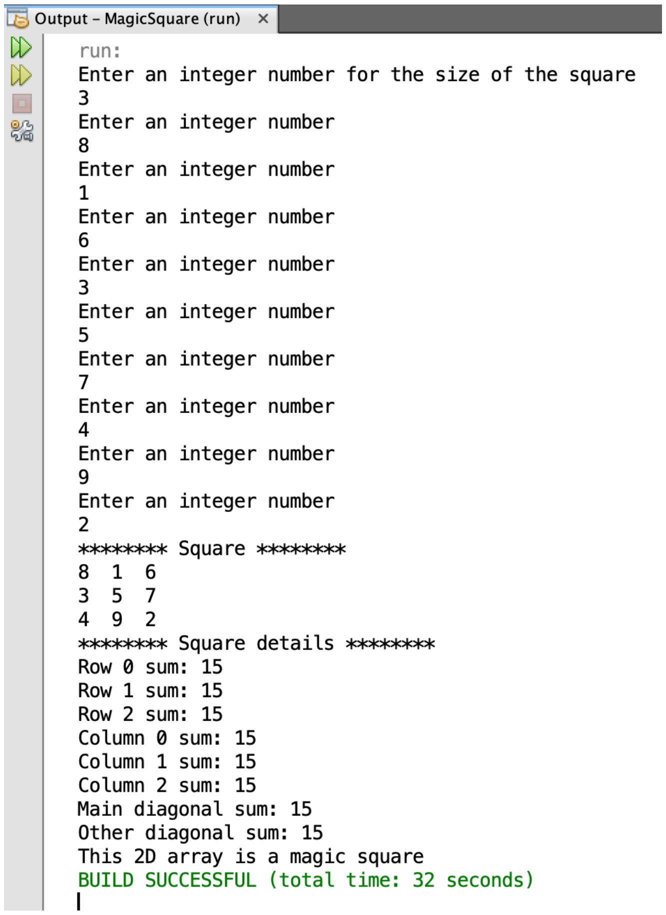
```
//---------------------------------------
//return true if the square is magic (all rows, cols, and diags have
//same sum), false otherwise
//---------------------------------------
public boolean isMagic() {
    //Add code here!

    return true;
}
```

2. Switch over to the main method to test your method. Delete the two lines that include “ >>>>>Delete this line to test part D”.



```
//Part D -
/* >>>>>Delete this line to test part D
//determine and print whether it is a magic square
if (sq.isMagic()) {
    System.out.println("This 2D array is a magic square");
} else {
    System.out.println("This 2D array is not a magic square");
}
>>>>>Delete this line to test part D */
```

3. Run your program to make sure that your method is working as expected. Fix any bugs you find and test your code with different values to make sure it works regardless of the square size and content. An example output might look like this:



```
run:
Enter an integer number for the size of the square
3
Enter an integer number
8
Enter an integer number
1
Enter an integer number
6
Enter an integer number
3
Enter an integer number
5
Enter an integer number
7
Enter an integer number
4
Enter an integer number
9
Enter an integer number
2
******** Square ********
8  1  6
3  5  7
4  9  2
******** Square details ********
Row 0 sum: 15
Row 1 sum: 15
Row 2 sum: 15
Column 0 sum: 15
Column 1 sum: 15
Column 2 sum: 15
Main diagonal sum: 15
Other diagonal sum: 15
This 2D array is a magic square
BUILD SUCCESSFUL (total time: 32 seconds)
```

4. When your program works and you are satisfied with the result, show your work to the instructional team to be checked off for this lab.

Bonus

- Create a method in the `Square` class that when called it randomly assigns values to the `Square` matrix. Hint: You might find the second constructor of the `Square` class useful!
- Create a method that checks the second condition for the magic square where all values must be unique

So what did you learn in this lab?

- How to write code to specifications (the methods you added to the Square class)
- Practiced testing and debugging a program