

## 1 Before Getting Started on the Exercises

For each of the following create a new project with an appropriate name and then write a program that solves the given problem. Remember to do the Shift+Ctrl+F to format the program and to put the standard comments at the top. For each program, you will be submitting the java code files through MyClasses, as you did before. **Note that since we are working with classes there will be multiple java files to upload, make sure you upload all the java files from the project.**

I also want either a Microsoft Word doc file (or LibreOffice Writer odt) or a text file (which you can create with NotePad++) of the output of at least three runs of the program that tests all of the program features. This doc (or odt) or text file is to be uploaded to MyClasses as well. You can copy and paste output from the Eclipse console area to the word or text program. Each program must include header comments with at least your name, date, and short description of the program.

### 1.1 Commenting

Up to now we have been doing minimal commenting (except for the projects). In general commenting your code is very important for both you and anyone who needs to read your code. When you are programming professionally or even if you are looking back at code you wrote in a previous class it is much easier if you have some comments on what the code is doing so that reading the details becomes clearer. So at this point we will put some rules on commenting your code. These are the minimum you need to do for this course, frankly you should be writing more than these.

- In the main program:
  - At the top of at least, your name, the date written, and a description of the program. As usual.
  - Each method should have a description of what it does. A couple lines for smaller methods but a more detailed description for larger ones. In this you should have a description of what the parameters are coming in and what the method is returning.
  - Major blocks of code should contain brief but descriptive comments to their function.
- For each class structure:
  - At the top have at least, your name, the date written, and a description of the class.
  - Each member variable should have a description.
  - Each member method should have a description of what it does. A couple lines for smaller methods but a more detailed description for larger ones. In this you should have a description of what the parameters are coming in and what the method is returning.
  - Major blocks of code should contain brief but descriptive comments to their function.

## 2 Exercises

1. This exercise uses the card and deck classes that we went over in class. The version we did in class is on the MyClasses page for this course, a different version can be found in the code examples given at the beginning of the course, in the blackjack example. Read through the card and deck classes to make sure you understand how each of the methods work. You may need to add methods to the classes to complete the assignment.

A *derangement* is when you have a list of objects and then mix them up so that no object is in the same position as it started. For example, if we have the list 1 2 3 4 5, then the list 2 5 4 1 3 is a derangement of the original list but 2 1 3 5 4 is not since 3 is in the same position as where it started.

The same can be done with larger lists and is a common game using a deck of cards. Take a new deck of cards out of its wrapper and then bet someone that no matter how many times they shuffle the deck there will be at least one card in the same position as it was in the originally manufactured deck. Question is, would you take that bet? Most people do since they feel that if they shuffle the deck enough times then the cards will all be mixed up.

In these printouts, the top row is the original order of the deck and the second row is the shuffled deck. Looking at the first run it is clear that this is not a derangement.

AH	2H	3H	4H	5H	6H	7H	8H	9H	10H	JH	QH	KH	AD	2D	3D	4D	5D	6D	7D	8D	9D	10D	JD	QD	KD
AS	6S	5C	9S	4H	6H	8H	10C	2H	9D	7C	AC	8S	3C	6D	4C	5H	QD	2D	5S	QC	3S	7H	2C	AD	KD

AC	2C	3C	4C	5C	6C	7C	8C	9C	10C	JC	QC	KC	AS	2S	3S	4S	5S	6S	7S	8S	9S	10S	JS	QS	KS
JD	10S	AH	JH	5D	4S	QH	2S	8D	JS	7S	KS	KC	9H	8C	3D	10D	6C	KH	9C	QS	4D	10H	JC	7D	3H

On the other hand, the following is a derangement,

AH	2H	3H	4H	5H	6H	7H	8H	9H	10H	JH	QH	KH	AD	2D	3D	4D	5D	6D	7D	8D	9D	10D	JD	QD	KD
10C	5S	3C	7H	KH	2S	10D	5D	JC	10S	AS	2H	QC	10H	JS	4S	7S	5H	4D	KD	AC	3D	KS	8D	9S	4C

AC	2C	3C	4C	5C	6C	7C	8C	9C	10C	JC	QC	KC	AS	2S	3S	4S	5S	6S	7S	8S	9S	10S	JS	QS	KS
KC	4H	3S	9H	9D	2C	6D	AD	JH	8C	JD	8H	QD	6C	2D	6H	AH	7D	3H	9C	QH	6S	8S	QS	5C	7C

Write a program that will simulate the shuffling many times and determine if the shuffle produces a derangement. We will count the number of derangements and then find the probability of a shuffled deck being a derangement. This program is not long if you use the methods in the card and deck classes in a smart way, in fact the program will probably be less than 40 lines in length. The program should take the number of trials to be done from the user. Make sure that they input a value between 100 and 1,000,000. Then for each trial you will create two decks of cards, shuffle one of them and do not shuffle the other. Then go through the two decks card by card, if there is a match of a pair of cards then the shuffle is not a derangement and if there are no matches then the shuffle is a derangement. Keep a count of the number of derangements found in the number of trials the user wanted. Then calculate the probability of a derangement by dividing the number of trials into the number derangements found. Also calculate one over the probability.

Specifically,

- Ask the user for the number of trials, make sure that they input a value between 100 and 1,000,000.
- For each trial,
  - Create two decks of cards, shuffle one of them and do not shuffle the other.
  - Go through the two decks card by card, to see if there are any card matches. If there is no match of any pair of cards between the two decks then the shuffle is a derangement. One way to do this is as follows.
    - Create a counter and set it to 0.
    - Use a for loop to go through the entire deck of cards, card by card. If there is a match between the two cards increment the counter.
    - At the end of the loop, if the counter is 0 the shuffle was a derangement and if the counter is greater than 0 there was a match and hence the shuffle was not a derangement.

- iii. If the shuffle is a derangement, increment a derangement counter.
- (c) Find the probability by dividing the number of derangements by the number of trials.
- (d) Calculate one over the probability as well.

Answer the following questions, put the answers in your runs file.

- (a) What is the approximate probability of a shuffled deck of cards being a derangement?
  - (b) What is the approximate value of one over the probability of a shuffled deck of cards being a derangement?
  - (c) Does the number for one over the probability look close to a very famous number you have seen before? If so, what is the famous number?
2. In this exercise you will be working with arrays of objects that you have created. You will construct a sphere class that will have several methods, including one that will take in another sphere as a parameter and return true if the two spheres collide and false if they do not.

These calculations lie at the heart of nearly all computer and video games. In most games, especially first-person-shooters, scores are increased when a bullet or laser beam hits (collides) with a target. Also, if the player runs into a wall they do not go through the wall (well usually) but instead bounce off of it or slide along it. Determining what objects hit each other is called *collision detection*. Collision detection is usually taken care of in a game's physics engine that is responsible for most object placement throughout the play of the game. In a physics engine, the objects in a scene are given physical attributes, such as shape, size, velocity, mass, momentum, bounce, frictions, rotational velocity, gravity, and even squishiness. The physics engine uses these attributes and time to determine how the objects interact with each other and where their new positions will be. As a part of that, the engine will determine which objects have collided and adjust their velocities and momentums accordingly. General collision detection is a very complex problem, but for simple objects like spheres (say in a billiards game) it is quite doable.

- (a) Create a Sphere class (object) that holds the center of the sphere and its radius. The center of the sphere will be three coordinates ( $x, y, z$ ) and the radius is just a decimal number. Make sure that the object has a constructor that sets all of the data in the Sphere class and accessor functions that can both get all the data from the object and set all of the data in the object. Have your accessor functions, as well as the constructor, make sure that the radius of the sphere is greater than or equal to 0. Add in methods for finding the volume and surface area of the sphere. Add in a method called `collide` that takes in a Sphere as a parameter and returns true if the calling sphere collides with the input sphere and false if they do not. The header to the method should look like the following.

```
public boolean collide(Sphere s)
```

Testing if two spheres collide is fairly easy, you calculate the distance between the centers of the two spheres and if this is less than or equal to the sum of the radii of the two spheres then the two are colliding. The distance between two points in three dimensions is

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

Finally, add a method that will print the sphere data to the screen, that is, the sphere's center, radius, volume and surface area. Recall that,

- Volume of a Sphere:  $V = \frac{4}{3}\pi r^3$

- Surface Area of a Sphere:  $A = 4\pi r^2$

(b) Write a main program that will ask the user three questions at the start,

```
Input the number of spheres (5-100): 20
Print Sphere Information (Y/N): y
Print Collision List (Y/N): y
```

The first is the number of spheres to create, the second is to print out the sphere information, and third to print out a collision list. Do data input checking on all of these inputs. If the number of spheres is not between 5 and 100 the program should ask for the input again until the user types in a number between 5 and 100. Same goes for the character inputs. Here you should take in a string and extract the first character. If the characters are not Y, y, N, or n then the program should ask for the input again. The answers to the last two questions are to be stored as single characters.

The program then creates an array of random spheres, as many as the user specifies. The centers should be randomly generated so that each coordinate is a double between -10 and 10. The radius of the sphere should be a random double between 1 and 3.

If the user selects to see the sphere information then the program should print out the center, radius, volume, and surface area of each sphere.

```
Center: (5.07972603700521, 8.382451098049518, 5.986081502426652)
Radius: 1.7061914719633906
Volume: 20.805200717552825
Surface Area: 36.581827525391425
```

If the user selects to see the sphere collision list, then the program should print out the listing of all collisions, as below. The list should list each collision only one time, so if spheres 3 and 14 collide it should list Spheres 3 and 14 and both Spheres 3 and 14 and Spheres 14 and 3.

```
Collision List
Spheres 1 and 19
Spheres 3 and 14
Spheres 3 and 16
Spheres 4 and 20
Spheres 7 and 8
Spheres 10 and 20
Spheres 11 and 19
Spheres 14 and 16
```

```
Number of Collisions: 8
```

A sample run of the program is below.

```
Input the number of spheres (5-100): 20
Print Sphere Information (Y/N): y
Print Collision List (Y/N): y

Center: (5.1756706654858355, 7.633939460837151, -9.199665632660894)
Radius: 1.6174420728196897
Volume: 17.72453362206822
Surface Area: 32.8751191524943

Center: (4.3419710151715485, 4.774029643662004, -8.062020845323566)
Radius: 2.619659902388656
Volume: 75.30491926804632
Surface Area: 86.23820122533672

Center: (-2.6782782051888576, 4.680873586759375, 7.763056149691117)
Radius: 1.2255716003190926
```

---

Volume: 7.710892975665975  
Surface Area: 18.87501221550422

Center: (-1.4122873246339296, 4.569617890179895, 5.190520533901752)  
Radius: 2.468399836935973  
Volume: 62.99921478058671  
Surface Area: 76.56686794160669

Center: (-9.585554991627818, 3.6460556794015417, -8.677103526759673)  
Radius: 2.0989299909362176  
Volume: 38.73311885979435  
Surface Area: 55.36123504888931

Center: (5.109987317811191, -5.247166587881951, 4.348219570636768)  
Radius: 1.2621021910379961  
Volume: 8.42116489467354  
Surface Area: 20.016996138199442

Center: (9.953691987660818, 7.132930829435896, -8.043845068336335)  
Radius: 2.7677831746819592  
Volume: 88.81468996271711  
Surface Area: 96.26623657713648

Center: (-2.991649805015724, -0.42629066851339203, -6.018110819893552)  
Radius: 2.0627114975471468  
Volume: 36.76241872280145  
Surface Area: 53.4671263041639

Center: (-0.786353229424062, -6.785627673491856, 8.038168761433447)  
Radius: 2.5438144382509007  
Volume: 68.95168651674798  
Surface Area: 81.3168824108394

Center: (4.031057101071333, -1.055794721818998, -0.42835648870361354)  
Radius: 1.9487972991736358  
Volume: 31.00192180342746  
Surface Area: 47.72469945936418

Center: (-3.601393793804535, -4.411509323441784, 1.0804552659556865)  
Radius: 2.218109163955339  
Volume: 45.712750955689664  
Surface Area: 61.826647261365466

Center: (-6.132802644630382, -8.026464691813935, 9.155785357826488)  
Radius: 1.4430920745372318  
Volume: 12.588405787287158  
Surface Area: 26.16965197731542

Center: (-1.4715044482167698, -1.5182377222773766, -7.4391295156131605)  
Radius: 1.2983240787505292  
Volume: 9.16722612576475  
Surface Area: 21.18244498997592

Center: (2.7076270618217713, 1.8523729075640922, 9.736764786670445)  
Radius: 2.701869749512645  
Volume: 82.61936183069881  
Surface Area: 91.73576392304045

Center: (8.3822498709795, 0.7525922553204651, -0.07509155028925463)  
Radius: 2.355353664807474  
Volume: 54.733975192015016  
Surface Area: 69.71433973142496

Center: (-6.097177238941307, -0.9609418001937833, -1.4776468148999715)  
Radius: 1.2864780599977519  
Volume: 8.918581007057133  
Surface Area: 20.797667564745066

Center: (5.827176767445337, 8.648445104417355, -5.663729776465156)  
Radius: 1.5498059137030316  
Volume: 15.592672255325759  
Surface Area: 30.18314509731618

Center: (8.545789970949006, 4.700930760493129, -5.919859954140668)  
Radius: 2.8373362813474605  
Volume: 95.67997291462163  
Surface Area: 101.16527978401936

Center: (9.862432752037307, 1.6696652400616294, 7.754825711082937)  
Radius: 2.908600478015007  
Volume: 103.07202839355142

```

Surface Area: 106.31095178519698
Center: (-3.191862059652788, 1.3639159880652052, -2.51651359085332)
Radius: 2.470711367085027
Volume: 63.17636719490505
Surface Area: 76.71033699428992

Collision List
Spheres 1 and 2
Spheres 2 and 18
Spheres 3 and 4
Spheres 7 and 18
Spheres 8 and 13
Spheres 8 and 20

Number of Collisions: 6

```

3. The transpose of a matrix/array is when you interchange the rows and columns.

-6	-10	-3	10	9
9	-2	7	-5	2
-4	2	-4	-8	-3

  

-6	9	-4
-10	-2	2
-3	7	-4
10	-5	-8
9	2	-3

Table 1: An Array and its Transpose

Write a program that will allow the user to input the number of rows and columns of a two-dimensional array of integers. Make sure that you do user input checking so that the number of rows and columns are both between 2 and 5. Create an array of the given size and have the user input the data for the array. The prompt for each data item should tell the user the row and column number of the entry they are inputting. Since most users will be more comfortable starting to count at 1 instead of 0 have the displayed positions start at 1. That is, if the user is inputting position (0, 3) of the array the prompt should be asking for row 1 and column 4. Now create a second array that is the transpose of the first and have the program print out both arrays.

4. For this exercise you are going to implement some of the features of a spreadsheet. Below is a description of the four methods you will be implementing and a start on the main program you will use. In general, you will ask the user to input the number of rows and columns of the array they want. You will create the array of the correct size and populate it with random numbers between -50 and 50. Then you will call the methods to do some calculations and print out the results. Finally, you will create another array and load the original array along with the row and column statistics you calculated in the methods.

(a) **public static void** PrintArray(**double** A[][][], **int** w, **int** d)

This method prints out a two-dimensional array using the printf command having a total formatted width of *w* and the number of decimal points to the right of the decimal *d*.

(b) **public static void** populate(**double** A[][][])

This method is to populate the array with random decimal numbers between -50 and 50.

(c) **public static double**[][] rowstats(**double** A[][][])

This method is to return a two-dimensional array with the same number of rows as A and two columns. The first column is to contain the average of the entries in that row

of A. So the average of the first row of A is to be in the first row and first column of the returned array. Second row average in the second row and first column, and so on. The second column is to contain the row standard deviation. Recall that the standard deviation of a list of numbers is defined to be

$$\sqrt{\frac{\sum_{i=1}^n (x_i - \mu)^2}{n-1}}$$

(d) **public static double[][] ColMaxMin(double A[][])**

This method is to return a two-dimensional array with the same number of columns as A and two rows. The first row is to contain the column minimum and the second row is to contain the column maximum.

Below is the main program, you are to fill in the portions between the <<< and >>> but the remainder of the main is to be unaltered.

```
public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);

    int rows = 0;
    int cols = 0;

<<< Insert the code to get the number of rows and columns from
the user making sure that each is at least two. >>>

    double A[][] = new double[rows][cols];

    populate(A);
    System.out.println();
    PrintArray(A, 8, 2);

    double[][] rowStatistics = rowstats(A);
    System.out.println();
    PrintArray(rowStatistics, 8, 2);

    double[][] columnStatistics = ColMaxMin(A);
    System.out.println();
    PrintArray(columnStatistics, 8, 2);

<<< Insert the code to create an array B with two more rows and
columns than A. Load the array A into B in the upper right,
load the row statistics in the two right-most columns, and
load the column statistics in the bottom two rows. >>>

    System.out.println();
    PrintArray(B, 8, 2);
}
```

### Program Run

```

Input the number of rows: 4
Input the number of columns: 7

-2.12   25.09   46.32   34.76   -9.94   -36.31   3.51
 13.63  -17.25   42.63   12.94   -3.41    47.97   5.14
-35.50  -38.77   25.74  -21.18   1.74    45.57  -11.50
-37.55   42.95  -46.11  -31.63  -15.57    3.12  -13.82

  8.76   28.52
 14.52   23.56
 -4.84   31.42
-14.09   30.10

-37.55  -38.77  -46.11  -31.63  -15.57  -36.31  -13.82
 13.63   42.95   46.32   34.76   1.74    47.97   5.14

-2.12   25.09   46.32   34.76   -9.94   -36.31   3.51   8.76   28.52
 13.63  -17.25   42.63   12.94   -3.41    47.97   5.14   14.52   23.56
-35.50  -38.77   25.74  -21.18   1.74    45.57  -11.50  -4.84   31.42
-37.55   42.95  -46.11  -31.63  -15.57    3.12  -13.82  -14.09   30.10
-37.55  -38.77  -46.11  -31.63  -15.57  -36.31  -13.82    0.00   0.00
 13.63   42.95   46.32   34.76   1.74    47.97   5.14   0.00   0.00

```

## 3 Challenge Exercise

As usual, the Challenge Exercise is optional and is for extra credit.

A Magic Square is an  $n \times n$  square array of numbers consisting of the distinct positive integers  $1, 2, \dots, n^2$  arranged such that the sum of the  $n$  numbers in any horizontal, vertical, or main diagonal line is always the same number. The number  $n$  is called the order of the magic square.

<table border="1" style="border-collapse: collapse; width: 100%; height: 100%;"> <tr><td style="width: 33.33%;">4</td><td style="width: 33.33%;">9</td><td style="width: 33.33%;">2</td></tr> <tr><td style="width: 33.33%;">3</td><td style="width: 33.33%;">5</td><td style="width: 33.33%;">7</td></tr> <tr><td style="width: 33.33%;">8</td><td style="width: 33.33%;">1</td><td style="width: 33.33%;">6</td></tr> </table>	4	9	2	3	5	7	8	1	6	<table border="1" style="border-collapse: collapse; width: 100%; height: 100%;"> <tr><td style="width: 25%; text-align: center;">4</td><td style="width: 25%; text-align: center;">9</td><td style="width: 25%; text-align: center;">2</td><td style="width: 25%; text-align: center;">15</td></tr> <tr><td style="text-align: center;">3</td><td style="text-align: center;">5</td><td style="text-align: center;">7</td><td style="text-align: center;">15</td></tr> <tr><td style="text-align: center;">8</td><td style="text-align: center;">1</td><td style="text-align: center;">6</td><td style="text-align: center;">15</td></tr> <tr><td style="text-align: center;">15</td><td style="text-align: center;">15</td><td style="text-align: center;">15</td><td style="text-align: center;">15</td></tr> </table>	4	9	2	15	3	5	7	15	8	1	6	15	15	15	15	15
4	9	2																								
3	5	7																								
8	1	6																								
4	9	2	15																							
3	5	7	15																							
8	1	6	15																							
15	15	15	15																							

Table 2: Magic Square of Order 3

The sum of each row, column or diagonal is known as the magic constant, and is,

$$M = \frac{n(n^2 + 1)}{2}$$

Kraitchik (1942) gives general techniques of constructing even and odd squares of order  $n$ . For  $n$  odd, a very straightforward technique known as the Siamese method can be used. It begins by placing a 1 in the center square of the top row, then incrementally placing subsequent numbers in the square one unit above and to the right. The counting is wrapped around, so that falling off the top returns on the bottom and falling off the right returns on the left. When a square is encountered that is already filled, the next number is instead placed below the previous one and the

method continues as before. While there are other ways to create these, your program must use this algorithm. For example,

0   1   0	0   1   0	0   1   0	0   1   0
0   0   0	0   0   0	3   0   0	3   0   0
0   0   0	0   0   2	0   0   2	4   0   2
0   1   0	0   1   6	0   1   6	8   1   6
3   5   0	3   5   0	3   5   7	3   5   7
4   0   2	4   0   2	4   0   2	4   9   2

Table 3: Magic Square of Order 3 Construction

Write a program that will take an odd integer between 3 and 25 from the user and then construct a magic square of that order. A start to the main program is below. You will need to fill in the Insert code portions and create two methods, `CreateMagic` and `IsMagic`. The method `CreateMagic` takes a two-dimensional array as its only parameter and fills it, as outlined above, to create a magic square. You may assume that the array is square and that the number of rows and columns is odd. The method `IsMagic` returns a boolean value and checks that the array forms a magic square.

```
public static void main(String[] args) {
    int n = 0;

    <<< Insert code to get an odd integer between 3 and 25 and store in the variable n >>>

    <<< Insert code to create an n X n integer array called MS >>>

    CreateMagic(MS);
    System.out.println();
    Print2DintArray(MS);
    System.out.println();

    if (IsMagic(MS))
        System.out.println("This is a magic square.");
    else
        System.out.println("This is not a magic square. OOPS!!!!");
}
```

Program Run											
Input the size of the Magic Square (odd number between 3-25): 11											
68	81	94	107	120	1	14	27	40	53	66	
80	93	106	119	11	13	26	39	52	65	67	
92	105	118	10	12	25	38	51	64	77	79	
104	117	9	22	24	37	50	63	76	78	91	
116	8	21	23	36	49	62	75	88	90	103	
7	20	33	35	48	61	74	87	89	102	115	
19	32	34	47	60	73	86	99	101	114	6	
31	44	46	59	72	85	98	100	113	5	18	
43	45	58	71	84	97	110	112	4	17	30	
55	57	70	83	96	109	111	3	16	29	42	
56	69	82	95	108	121	2	15	28	41	54	
This is a magic square.											