1 Exercise

In this homework you will be working with ArrayLists of objects. You may get a deja vu feeling in this homework because it does the same thing we did before with the sphere collisions except the storage structures are different. Instead of storing the information about the spheres and collisions in arrays you will be storing them in Java's ArrayList structure. Remember that with an ArrayList you do not need to specify the size of the storage when you create it, these structures automatically get larger if they need to. Also, the Sphere class from the previous homework will not need to be altered, if you created a working version in the previous homework you may use it and if not you may use the one I created and is on the MyClasses page for this assignment.

As usual, you will submit the Java code files for main program and the Sphere class, and either a Word, LibreOffice, or text file containing at least three runs of the program.

If you use your own Sphere class make sure that it has all the needed functionality. We will go over its functionality again here for completeness.

Make sure you have a working Sphere class (object) that holds the center of the sphere and its radius. The center of the sphere will be three decimal 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. Have methods for finding the volume and surface area of the sphere. Also make sure that there is a working version of 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. 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, make sure you have a method that will print the sphere data to the screen, that is, the sphere's center, radius, volume and surface area. Make sure that each method of the Sphere class is commented with a description of the method, the input parameters and what they are, and the output of method and what it is, if there is an output to the method.

- Volume of a Sphere: $V = \frac{4}{3}\pi r^3$
- Surface Area of a Sphere: $A = 4\pi r^2$

Write a main program that will ask the user four questions at the start,

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

The first is the number of spheres to create, the second is to print out the sphere information, third to print out a collision list, and a maximum 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 three questions are to be stored as single characters.

The program then creates an ArrayList 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.

The program should then find all the spheres that collide. If it finds a collision the program should store all of the collision information in an ArrayList. Really all that needs to be stored here is the number of spheres, so if spheres 4 and 11 collide the ArrayList should contain the numbers 4 and 11. In the previous homework you simply printed these out to the screen but here you will be storing them since you will need to find the maximum number of sphere collisions.

The program should then find all the spheres that have the maximum number of collisions. That is, for each sphere find the number of spheres it collides with, and then find the maximum of these numbers. Furthermore, find all the spheres that collide the maximum number of times. It is very possible that there are several spheres that collide with the same number of other spheres. All of these maximum collision spheres are to be stored in another ArrayList.

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 4 and 11 collide it should list 4 - 11 and not both 4 - 11 and 11 - 4.

```
Number of Collisions: 6
Collision List
1.  0 - 3
2.  4 - 5
3.  4 - 11
4.  8 - 15
5.  8 - 18
6.  13 - 16
```

If the user selects to see the maximum sphere collision list, then the program should print out the listing of all spheres that have the maximum number of collisions, as below.

```
Maximum Number of Collisions: 2
Spheres with 2 Collisions: 4  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
Print Maximum Collision List (Y/N): y

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

Center: (-6.807076495311934, 3.224892834110493, -5.376386237794126)
Radius: 2.8159353196165737
Volume: 93.53122761664173
Surface Area: 99.64493179059657

Center: (-4.069674646015368, -0.1361157419393635, -9.411059601111408)
Radius: 1.2482215489329997
Volume: 8.146360722650074
Surface Area: 19.579122142892942

Center: (7.8584397744780645, 8.962312545185075, 3.606660565315547)
Radius: 2.6985009167537015
Volume: 82.31070455419187
Surface Area: 91.50714462592627

Center: (3.8234369820430576, -8.872560952838334, -6.749469836089799)
Radius: 2.5649029725879666
Volume: 70.68079601806485
Surface Area: 82.6707249047497

Center: (2.5218882947438015, -8.314814817798702, -8.843475324903586)
Radius: 1.6041130603290268
Volume: 17.28994188763025
Surface Area: 32.33551733083671

Center: (4.371071239265882, 3.7934384233761165, 6.733765855247956)
Radius: 1.768960225727202
Volume: 23.18690665286731
Surface Area: 39.32294177502279

Center: (-8.01732638171239, 9.609345339706167, 6.757231133935978)
Radius: 1.5400336731033553
Volume: 15.29957122988578
Surface Area: 29.80370786124816

Center: (6.485130781149195, -1.9267783344186498, -1.7699428975819504)
Radius: 2.604659519430519
Volume: 74.0187063778793
Surface Area: 85.2534150729183

Center: (-8.993751798266436, 4.844346401888524, 6.032163408544907)
Radius: 1.0900863593451413
Volume: 5.425894246355314
Surface Area: 14.932470807950116

Center: (3.819157135610764, 5.021585936088712, 2.125824596738461)
Radius: 2.7237433489418272
Volume: 84.64224183740575
Surface Area: 93.22711172874187

Center: (7.971059471936552, -7.495571914334911, -8.532269712608933)
Radius: 2.2719502851238715
Volume: 49.123022529014015
Surface Area: 64.86456528207314

Center: (1.655634145962864, 1.1976821406433498, -8.719026754022698)
Radius: 1.3994746961636702
Volume: 11.48110689711196
Surface Area: 24.61160661622116

Center: (4.156364084387514, 2.7357422182571174, -5.030690280134593)
Radius: 1.6958461189605507
Volume: 20.429038872763154
Surface Area: 36.1395505954601

Center: (-4.120528558260461, -7.000022551039493, 9.70995913921793)
Radius: 1.0303548140475975
Volume: 4.581936044934381
```

```
Surface Area: 13.340849139923709

Center: (7.81153700110875, -4.574538631420319, -2.3599102072806595)
Radius: 1.6261756174616138
Volume: 18.013203047226117
Surface Area: 33.23110281657754

Center: (5.678294410032107, 2.940917179000l416, -5.443932127475568)
Radius: 1.285075674592516
Volume: 8.889446444183553
Surface Area: 20.75234934394577

Center: (-2.8525004862398102, -7.641599311287573, 7.17343903474805)
Radius: 1.047678027536257
Volume: 4.816949758177592
Surface Area: 13.793215944898371

Center: (6.610167214492556, -4.87308862789307, 0.24952603282191)
Radius: 1.2402708220204206
Volume: 7.991681915385017
Surface Area: 19.330492438013927

Center: (-8.609106011537959, -3.4132291207487286, -1.7505765859612374)
Radius: 1.337665679585055
Volume: 10.026084555330254
Surface Area: 22.485628602896544

Number of Collisions: 6
Collision List
1.  0 - 3
2.  4 - 5
3.  4 - 11
4.  8 - 15
5.  8 - 18
6.  13 - 16

Maximum Number of Collisions: 2
Spheres with 2 Collisions: 4  8
```

2 Challenge Exercise

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

The Complex Number Class: This exercise is to create a new class to handle complex numbers. As a quick review, a complex number is of the form $a + bi$ where a and b are real numbers and $i = \sqrt{-1}$. The value a is called the real part and the value b is called the imaginary part. To add two complex numbers you simply add the real parts and the imaginary parts, specifically,

$$(a + bi) + (c + di) = (a + c) + (b + d)i$$

Subtraction is similar,

$$(a + bi) - (c + di) = (a - c) + (b - d)i$$

For multiplication you simply distribute all products and combine like terms at the end, remembering that $i^2 = -1$, specifically,

$$(a + bi)(c + di) = ac + adi + bci + bdi^2 = ac + adi + bci - bd = (ac - bd) + (ad + bc)i$$

Division of complex numbers requires the use of the complex conjugate. The conjugate of the complex number $a + bi$ is $a - bi$. Now to do division we take the conjugate of the denominator and multiply by the conjugate over itself (that is by 1),

$$\frac{a + bi}{c + di} = \frac{a + bi}{c + di} \cdot \frac{c - di}{c - di} = \frac{(a + bi)(c - di)}{(c + di)(c - di)} = \frac{ac + bd + (bc - ad)i}{c^2 + d^2} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}i$$

As long as the denominator is not 0 we have a legitimate calculation.

Create a complex number class called ComplexNumber. It should have private data members named real and imag for the real and imaginary parts, these should both be doubles. Include the following functionality,

- `ComplexNumber()`

  A default constructor that sets the number to 0.

- `ComplexNumber(double r, double i)`

  Constructor that sets r to the real part and i to the imaginary part.

- `ComplexNumber add(ComplexNumber a)`

  Method to add two complex numbers, so the command `c.add(d)` will return the complex number $c + d$.

- `ComplexNumber subtract(ComplexNumber a)`

  Method to subtract two complex numbers, so the command `c.subtract(d)` will return the complex number $c - d$.

- `ComplexNumber multiply(ComplexNumber a)`

  Method to multiply two complex numbers, so the command `c.multiply(d)` will return the complex number $c \cdot d$.

- `ComplexNumber divide(ComplexNumber a)`

  Method to divide two complex numbers, so the command `c.divide(d)` will return the complex number c/d.

- `double modulus()`

  Method that returns the modulus of a complex number, so the command `c.modulus()` will return the modulus of the complex number c. If $c = a + b \cdot i$ then its modulus is defined as $\sqrt{a^2 + b^2}$.

- `String toString()`

  Method that returns a string representing the real and imaginary parts of the complex number, for example `-4.0 + 10.0i` and `0.44828 + -0.3793i` would be acceptable formats.

Make sure you test your class extensively with a main program. For example, with the following program

```
public class HW_11_EC_ComplexTester {

    public static void main(String[] args) {
        ComplexNumber a = new ComplexNumber(1, 3);
        ComplexNumber b = new ComplexNumber(-2, 5);
        ComplexNumber c = new ComplexNumber(3, -4);

        System.out.println(a.add(b));
        System.out.println(a.subtract(b));
        System.out.println(a.multiply(b));
        System.out.println(a.divide(b));
        System.out.println(a.modulus());
        System.out.println();

        System.out.println(b.add(b));
        System.out.println(b.subtract(b));
        System.out.println(b.multiply(b));
        System.out.println(b.divide(b));
        System.out.println(b.modulus());
        System.out.println();

        System.out.println(b.add(c));
        System.out.println(b.subtract(c));
        System.out.println(b.multiply(c));
        System.out.println(b.divide(c));
        System.out.println(c.modulus());
        System.out.println();
    }
}
```

the output should be

```
-1.0 + 8.0i
3.0 + -2.0i
-17.0 + -1.0i
0.4482758620689655 + -0.3793103448275862i
3.1622776601683795

-4.0 + 10.0i
0.0 + 0.0i
-21.0 + -20.0i
1.0 + 0.0i
5.385164807134504

1.0 + 1.0i
-5.0 + 9.0i
14.0 + 23.0i
-1.04 + 0.28i
5.0
```

Once your ComplexNumber class is working create a new java project and a new main for that project. Copy over the ComplexNumber class file to your new project. Copy the following method over to your new project. This method takes as parameters a two-dimensional array of Color objects and a string holding a filename. It will create an image that is the same size as the array (in pixels) and each pixel on the image will have the color stored in the respective location.

```java
public static void exportImage(Color A[][], String filename) {
    int rows = A.length;
    int cols = A[0].length;

    BufferedImage bi = new BufferedImage(cols, rows, BufferedImage.TYPE_INT_RGB);

    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            bi.setRGB(j, i, A[i][j].getRGB());

    try {
        File outputfile = new File(filename);
        ImageIO.write(bi, "png", outputfile);
    } catch (Exception e) {
    }
}
```

So if the color in the $(302, 421)$ position of the array is green then the pixel in the $(302, 421)$ pixel position of the image is green. Although the Color class that is built into java has a lot of features we will only need to use one of its constructors. To create a color you use the syntax

```
new Color(r, g, b);
```

Where r, g, and b are the amounts of red, green, and blue that are being used to create the color. Each of these must be in the range of 0 to 255, where 0 is no color and 255 is all color. So an (r, b, g) of $(0, 0, 0)$ is black, an (r, b, g) of $(255, 255, 255)$ is white, an (r, b, g) of $(255, 0, 0)$ is bright red, and so on.

To use this method and Java's Color class you need to include the following at the top of the program.

```
import java.awt.Color;
import java.awt.image.BufferedImage;
import java.io.File;
import javax.imageio.ImageIO;
```

Write a main program that does the following. Allow the user to input 5 values,

1. The image width and height, an integer.
2. The maximum iteration, an integer.
3. The real part of a complex number c, a decimal.
4. The imaginary part of a complex number c, a decimal.
5. The color power, a decimal.

For example, the user would have something like the following come up on the screen. They, of course, typed in the 1000, 500, -.194, .6557, and 0.5.

```
Image width and height: 1000
Maximum Iteration: 500
Real part of c: -.194
Imaginary part of c: .6557
Color power: 0.5
```

To make this algorithm easier to discuss lets say that the image height and width is `size`, the maximum iteration is `m`, the real part of c is `r`, the imaginary part of c is `i`, and the color power is p.

Now have the program do the following,

1. Create a two-dimensional array of color objects with the number of rows and columns both being `size`.
2. Create a new complex number c with real and imaginary parts of r and i respectively.
3. For each entry in the two-dimensional array do the following. We will designate the row position in the array as `row` and the column position in the array as `col`.
   (a) Initialize a counter (call it `count`) to 0.

(b) Set the value of x to $3 \cdot \frac{col}{size} - 1.5$.

(c) Set the value of y to $1.5 - 3 \cdot \frac{row}{size}$.

(d) Create a new complex number z with real part of x and imaginary part of y.

(e) While the modulus of z is less than 4 and the counter is less than the maximum iteration selected by the user do the following,

   i. Replace z by $z^2 + c$.

   ii. Increment the counter.

(f) If the counter equals the maximum iteration put the color black into the `row` and `col` position of the array.
If not, calculate $t = \left(\sin\left(\frac{\pi}{2} \cdot \frac{c}{m}\right)\right)^p \cdot 255$, where c is the counter value, m is the maximum iteration selected by the user, and p is the color power selected by the user. Convert this value to an integer (we will still call it t) and then put the color $(t, 0, 0)$ into the `row` and `col` position of the array. Note that you can do the sine function using `Math.sin`.

4. At this point, each cell of the array will have a color in it. Send the array and a filename (something like `MyPicture.png`) to the exportImage method.

Some runs you might want to do are the following. Look at the image after each run.

```
Image width and height: 1000
Maximum Iteration: 500
Real part of c: -.194
Imaginary part of c: .6557
Color power: 0.5

Image width and height: 1000
Maximum Iteration: 25
Real part of c: 0
Imaginary part of c: 1
Color power: 1

Image width and height: 1000
Maximum Iteration: 1000
Real part of c: -.74543
Imaginary part of c: .11301
Color power: .5

Image width and height: 1000
Maximum Iteration: 2000
Real part of c: .27664
Imaginary part of c: .00742
Color power: .35

Image width and height: 1000
Maximum Iteration: 100
Real part of c: .11031
Imaginary part of c: -.67037
Color power: .5
```