CPS109 Lab 7

Source: Big Java, Chapter 7
Preparation: read Chapter 7 and the lecture notes for this week.

Objectives:

1. **To practice using one- and two-dimensional arrays**
2. **To practice using partially filled arrays**

Instructions:
The lecture notes for week 7 indicate that you should do the following exercises from Big Java: Exercises P7.2, .3, .4, .8, .12 and Project 7.1. In this lab we lead you through four of these exercises, and two graphical exercises.

Append your work (the .java files you develop for each exercise) to a file called **lab7.txt** and submit it **on Blackboard** by 11:59, Saturday, Oct. 31**.** Note, rather than emailing your TA the solution, you should upload **lab7.txt to Blackboard.** To do this, find the Assignments folder on my.ryerson.ca, click on lab7, and when you are ready, click on submit to upload your file. Make sure that when you are done, you click on **submit**, rather than save, since save means you are still working on it. The TA cannot see it until you press submit. You can consult with your colleagues and the TA regarding how to do the lab, but what you submit must be your own **individual** work. Academic integrity is taken very seriously at Ryerson. See the course management form for more information.

---

1. **Exercise P7.2, 7.3 and 7.4 (combined and modified).**

Implement a class Purse. A purse contains a collection of coins. For simplicity we will only store the coin names in the collection. The book suggests that you use an ArrayList<String> to hold the collection, but we will focus on arrays for now and use an array of Strings to hold the collection. We will assume that the purse has a fixed size, which is the maximum number of coins that it can hold. Since this is a combination of three exercises all dealing with the Purse class, you should probably develop the solution in parts. You can add dummy return values to methods you have not yet written, so that the Purse class will compile and you can test the parts you are working on.

a) Supply a default constructor which sets the size of the purse to a default value, say 100. Supply another constructor which sets the size to the parameter.
b) Supply a method **boolean addCoin(String coinName)** which returns true if the coin was successfully added to the purse, and false otherwise. The coin is not added to the purse if the purse is full or if the coin is not a valid coin. Valid coins have names Nickel, Dime, Quarter, and Loonie. Use a private helper method **boolean legal(String coinName)** to test the legality of a name.
c) Add a method **reverse()** which reverses the sequence of the coins in the purse. Also add a method **toString** that prints the coins in the purse in the format **Purse[Dime,Nickel,Quarter,Dime]**.
d) Add a method **public void transfer(Purse other)** which transfers the contents of one purse to another. To do this, also add a method **String remove()** which removes

and returns the top coin of the purse. If there are no coins in the purse, then **remove()** returns **null.** Thus, if purse1 is **Purse[Quarter,Dime,Nickel]** and purse2 is **Purse[Nickel,Dime]** then after **purse1.transfer(purse2)** purse1 is **Purse[Quarter,Dime,Nickel,Dime,Nickel].**

e) The **PurseTester** class is given below, along with a **Tester** class which has a couple of methods for comparing values. The **Tester** methods are an aid to spotting errors, since they only print something when there is a mismatch. An outline of the **Purse** class which you are to fill in is also given.

```
/**
   This class contains some static methods to compare values.
 */
public class Tester
{
    /**
       Compares two boolean values. Prints a message if they are not equal.
       @param number the number of the test
       @param obtained the typically unknown value
       @param expected the expected value
     */
    public static void test(int number, boolean obtained, boolean expected)
    {
      if (obtained != expected) {
          System.out.println(number + ". Obtained: " + obtained) ;
          System.out.println(number + ". Expected: " + expected) ;
          System.out.println("-------------------------------------") ;
      }
    }
    /**
       Compares two String values. Prints a message if they are not equal.
       @param number the number of the test
       @param obtained the typically unknown value
       @param expected the expected value
     */
    public static void test(int number, String obtained, String expected)
    {
      if (!obtained.equals(expected)) {
          System.out.println(number + ". Obtained: " + obtained) ;
          System.out.println(number + ". Expected: " + expected) ;
          System.out.println("-------------------------------------") ;
      }
    }
}

/**
   A class to test the Purse class.
*/
public class PurseTester
{
   /**
      Tests the methods of the Purse class.
      @param args not used
   */
   public static void main(String[] args)
```

```
   {
      Purse harrysPurse = new Purse();
      Tester.test(1, harrysPurse.addCoin("Dime"), true);
      Tester.test(2, harrysPurse.addCoin("Nickel"), true);
      harrysPurse.addCoin("Quarter");
      Tester.test(3, harrysPurse.addCoin("Yen"), false);
      harrysPurse.addCoin("Dime");
      Tester.test(4, harrysPurse.toString(),
              "Purse[Dime,Nickel,Quarter,Dime]");
      harrysPurse.reverse() ;
      Tester.test(5, harrysPurse.toString(),
              "Purse[Dime,Quarter,Nickel,Dime]");
      Purse marysPurse = new Purse(2) ;
      marysPurse.addCoin("Dime") ;
      marysPurse.addCoin("Nickel") ;
      harrysPurse.transfer(marysPurse) ;
      Tester.test(6, harrysPurse.toString(),
              "Purse[Dime,Quarter,Nickel,Dime,Nickel,Dime]");
      Tester.test(7, marysPurse.toString(),
              "Purse[]");
   }
}

/**
   A purse has an array list of coin names that can
   be changed by adding coins.
*/
public class Purse
{
    //instance variables
    private String[] coins; //the money in the purse
    private int count ; // the number of coins in the purse
    private int size ; // the size of the purse
    private final int SIZE = 100 ; //default size

    /**
       Constructs a purse with a default size.
    */
    public Purse()
    {
      todo
    }

    /**
       Constructs a purse with a given size.
       @param size the size of the purse
    */
    public Purse(int size)
    {
      todo
    }

    /**
       Deposits money into the purse.
       @param coin the coin to deposit
```

```
    */
    public boolean addCoin(String coin)
    {
      todo
    }

    /**
       Private helper method to check legality of coin
       @return true if legal, false otherwise
    */
    private boolean legal(String coin)
    {
      todo
    }
    /**
       Produces string representation of money in purse.
       @return Purse[coin,coin,...]
    */
    public String toString()
    {
      todo
    }
    /**
       Reverses the order of the money in the purse
    */
    public void reverse()
    {
      todo
    }
    /**
       Removes top coin from the purse
       @return name of the coin removed, null if no coin removed
    */
    public String remove()
    {
      todo
    }
    /**
       Transfers all the money from a given purse to this purse
       @param other the reference to the other purse
    */
    public void transfer(Purse other)
    {
      todo
    }
}
```

---

2. Exercise 7.8

Write a program that reads a sequence of integers into an array and computes the alternating sum of all elements in the array. For example, if the program is executed with the input data

1 4 9 16 9 7 4 9 11
then it computes
1-4+9-16+9-7+4-9+11
and prints the value 2.

To solve this problem, follow the outline given in the following class. You will also have to add one more method to the **Tester** class (but you do not have to append Tester.java to your lab, just **Main**.java below). Note that when you are entering values from the keyboard in a unix-like environment, you indicate that there are no more values by pressing **Control**-**D**. Also note the simle way of copying an array in Java, and the important method of increasing the size of an array when it becomes full. These operations are discussed in Section 7.7 of Big Java.

```
/**
   Solution to Exercise 7.8
 */
import java.util.* ;

public class Main
{
    public static void main(String[] args)
    {
      final int MAX = 5 ;
      int[] values = new int[MAX] ;
      System.out.println("Enter a sequence of integers:") ;
      Scanner scanner = new Scanner(System.in) ;
      int count = 0 ;
      //read the values into the array
      while (scanner.hasNextInt()) {
          if (count == values.length) {
              //increase size of array
            int[] temp = new int[values.length * 2] ;
            System.arraycopy(values, 0, temp, 0, values.length) ;
            values = temp ;
          }
          values[todo] = scanner.nextInt() ;
          count++ ;
      }
      //Compute the alternating sum
      System.out.println("Alternating sum = " +
                     alternatingSum(values, count)) ;
      //Some automatic testing
      Tester.test(1, alternatingSum(new int[]{1, 2, 3}, 3), 2) ;
      Tester.test(2, alternatingSum(new int[]{}, 0), 0) ;
      Tester.test(3, alternatingSum(new int[]{1, 2, 3, 4, 5, 6}, 6), -3) ;
    }
    /**
       Computes the alternating sum of the values in the array.
       @param array the array of integers
       @param count the number of values in the array
       @return the alternating sum: first - second + third - ...
     */
```

```
    public static int alternatingSum(int[] array, int count)
    {
      todo
    }
}
```

---

3. Exercise P7.12 Magic squares

An n x n matrix that is filled with the numbers 1, 2, 3, ..., n^2 is a magic square if the sum of the elements in each row, in each column, and in the two diagonals is the same value. For example,

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

is a magic square. Write a class **Square** which can test if a sequence of numbers corresponds to a magic square. Below is the **SquareTester** class, which should run silently.

```
/**
   Tests the Square class of Exercise 7.12
 */
import java.util.* ;

public class SquareTester
{
    public static void main(String[] args)
    {
      Square square = new Square(new int[]{1, 2, 3}) ;
      Tester.test(1, square.isMagic(), false) ;
      square = new Square(new int[]{1,2,3,4,5,6,7,8,9}) ;
      Tester.test(2, square.isMagic(), false) ;
      square = new Square(new int[]{1}) ;
      Tester.test(3, square.isMagic(), true) ;
      square = new Square(new int[]
               {16,3,2,13,5,10,11,8,9,6,7,12,4,15,14,1}) ;
      Tester.test(4, square.isMagic(), true) ;
      square = new Square(new int[]{1,4,3,2}) ;
      Tester.test(5, square.isMagic(), false) ;
    }
}
```

As you can see, the constructor takes a one-dimensional array. The constructor checks the array to see that (1) it comprises n^2 numbers for some n, and (2) that each of the values 1, 2, 3, ..., n^2 appear exactly once. If the array does not meet these basic requirements, then a boolean instance variable **inputOK** is set to false. Otherwise the corresponding two-dimentional array is filled with the numbers. A method isMagic checks whether the array corresponds to a magic. Fill in the following Square class.

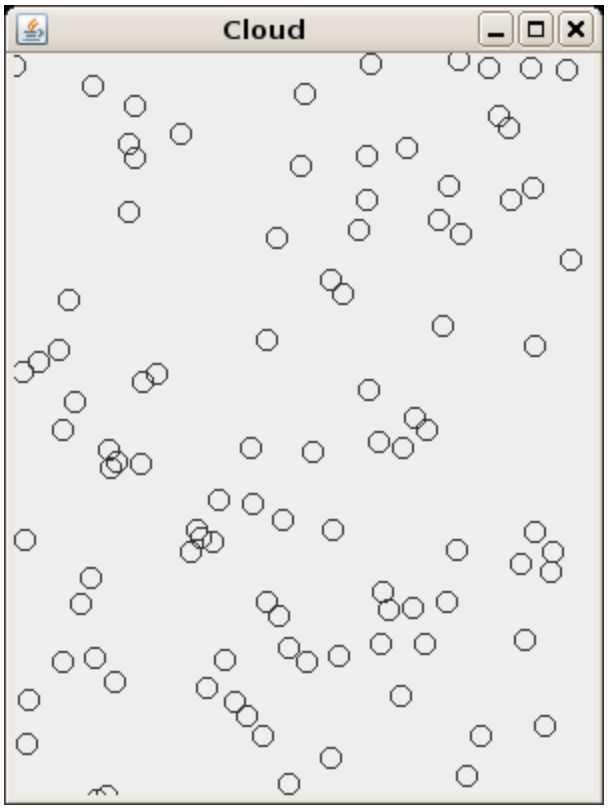
```
/**
   A square has a two-dimensional array of integers,
   1,2,3,...,n^2
   It can test itself for being a magic square, meaning on for
   which tue sum of each column equals that of each row and diagonal.
*/
public class Square
{
    //instance variables
    private int[][] square ; //the square
    private boolean inputOK = true ;

    /**
       Constructs a square from a linear array
       and checks that there are n^2 values and that each number from
       1 to n occurs exactly once.
       @param array the input array
    */
    public Square(int[] array)
    {
      //check squareness
      int n = (int)Math.sqrt(array.length) ;
      if (n * n != array.length)
          inputOK = false ;
      //check 1 to n values present
      else {
        todo
      }
      //put the numbers into the square if the input is OK
      if (inputOK) {
          square = new int[n][n] ;
          todo
      }
    }

    /**
       Tests the square for magic property
       @return true if rows, diagonals and columns each add to same
    */
    public boolean isMagic()
    {
      //if the input was not OK, return false right now.
      if (!inputOK)
          return false ;

      int n = square[0].length ;
      //get sum along one diagonal
      int diagSum = 0 ;
      for (int i = 0 ; i < n ; i++) {
          diagSum += square[i][i] ;
      }
      //Test if other sums are equal to diagSum
      todo
    }
}
```

4. Exercise P7.14

Implement a class **Cloud** that contains an array of Point2D.Double objects. Support methods

public void **add**(Point2D.Double aPoint)
public void **draw**(Graphics2D g2)

Draw each point as a tiny circle. Write a graphical application that draws a cloud of 100 random points.




```
import javax.swing.JFrame;
/**
   To display the cloud component.
*/
public class CloudViewer
{
```

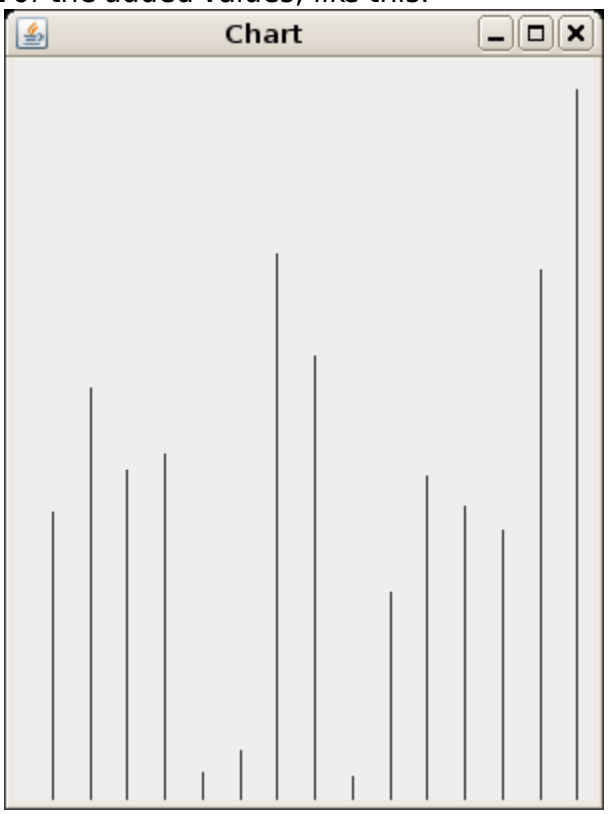
```
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();
      final int WIDTH = 300, HEIGHT = 400 ;
      frame.setSize(WIDTH, HEIGHT);
      frame.setTitle("Cloud");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      CloudComponent component = new CloudComponent(100, WIDTH, HEIGHT) ;
      frame.add(component);

      frame.setVisible(true);
   }
}

import javax.swing.JComponent;
import java.awt.geom.Ellipse2D ;
import java.awt.geom.Point2D ;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.util.Random;

/**
   A CloudComponent draws a number of random circles.
*/
public class CloudComponent extends JComponent
{
    //instance variables
    private Cloud cloud ; //the cloud to draw
    private Random random ; // generator

    /**
       Constructs a CloudComponent that contains a cloud with a given number
       of circles.
       @param n the number of circles to draw.
       @param width the width of the component initially
       @param height the height of the component initially
    */
    public CloudComponent(int n, int width, int height)
    {
      random = new Random() ;
      cloud = new Cloud() ;
      todo (add n random points to the cloud)
    }
    /**
       Draws the circles
       @param g the graphics context
     */
    public void paintComponent(Graphics g)
    {
      Graphics2D g2 = (Graphics2D) g ;
      cloud.draw(g2) ;
    }
}
import java.awt.Graphics2D;
```

```
import java.awt.Color ;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Line2D;
import java.awt.geom.Point2D;

/**
   A cloud shape that can add points to itself
   and draw itself
*/
public class Cloud
{
    //instance constants
    private final int RADIUS = 5 ; //radius of little circles
    private final int MAX = 100 ; //initial size of the array points
    //instance variables
    private Point2D.Double[] points ; //array of points
    private int count ; //number of points
    /**
       Constructs an empty cloud of points
    */
    public Cloud()
    {
      points = todo
      count = 0 ;
    }

    /**
       Adds a point to the cloud.
       @param point point to add
    */
    public void add(Point2D.Double point)
    {
      //increase the size of array if full
      if (count == points.length) {
          todo(as in previous exercise, double size and transfer)
      }
      points[count] = todo ;
      todo ;
    }
    /**
       Draws the cloud.
       @param g2 the graphics context
    */
    public void draw(Graphics2D g2)
    {
      for (int i = 0 ; i < count ; i++) {
          todo(make circle for each point specifying the centre)
          g2.draw(circle) ;
      }
    }
}
```

5. Exercise P7.16

Write a class **Chart** with methods
public void **add**(int value)
public void **draw**(Graphics2D g2)

that displays a stick chart of the added values, like this:



You may assume that the values are the heights of the lines in pixels. Also write the associated ChartViewer and ChartComponent classes, as in the previous problem.

The End