NON-LINEAR DATA STRUCTURES AND ALGORITHMS

LABS: Magic Square

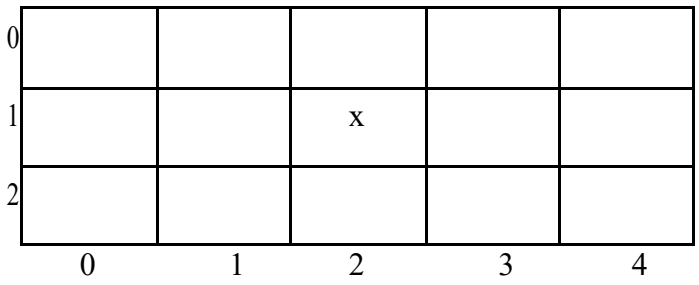
Part1: Learning to use multi-dimensional arrays. **(Week 2)**

PREVIOUS STUDY.

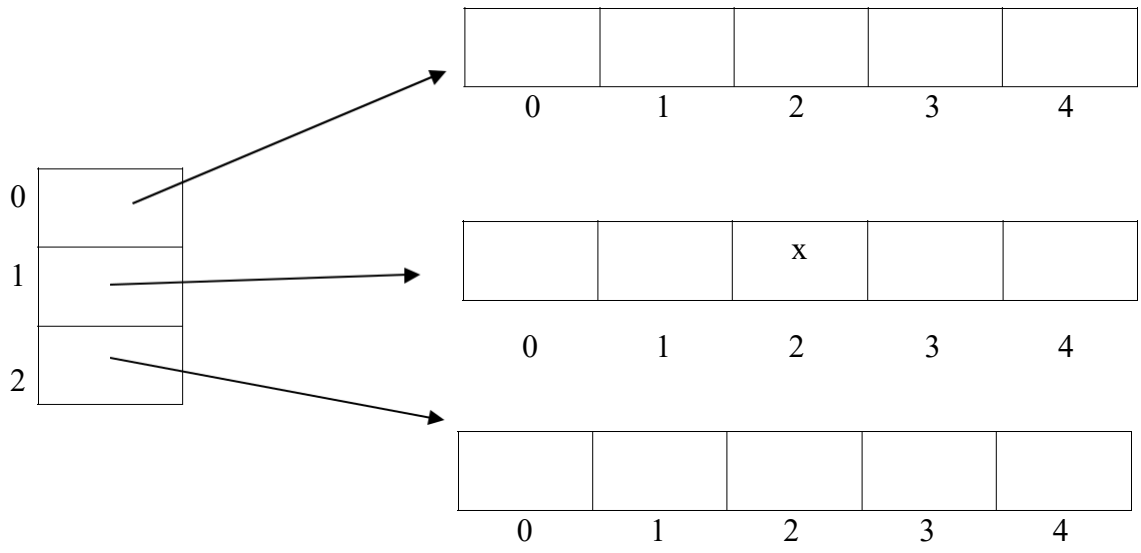
I. Concepts:

In Java, a Two-Dimensional array is implemented as a One-Dimensional array of One-Dimensional arrays. I.e., a list where each element points to another list. We can visualize a 2D array (also known as matrix) as a table, with rows and columns. I.e. an "*m by n*" 2D array can be seen as a table with *m* rows and *n* columns. But it's really a 1D array of *m* elements where each element points to a 1D array of *n* elements.

For example, here is how we see a "3 by 5" 2D array:



And here is how 2D arrays are implemented:



A Three-Dimensional array is a One-Dimensional array of Two-Dimensional arrays, etc. We can visualize it as a book, where each page contains a table.

II. Declaring 2D Arrays

type [][] *name* = **new** *type*[rows][cols] ;

1. *type* is the type of data stored in the array (any primitive type or class)
2. *name* is the name of the array object variable
3. *rows* is an integer expression indicating the number of rows
4. *cols* is an integer expression indicating the number of columns

Example**:**int [][] matrix = **new** int [5][4] ;

(creates a 2D array of 5 rows and 4 columns pointed to by *matrix*)

Naturally, the declaration of the object-variable and the creation of the object may be done separately:

```
int [][] matrix ;
…..
matrix = new int [5][4] ;
```

III. Accessing the Individual Elements of a 2D Array

This is similar to accessing the elements of a 1D array, but each element requires two indices.

***name*[*row*][*col*]**

- *name* is the name of the array object variable
- *row* is an integer expression that tells you which row (i.e. the index of the element of the 1D array that points to the other arrays)

- *col* is an integer expression that tells you which column (i.e. the index of the element in the array pointed to by *row*)

☞

The first index is *always* the "row" index

Example: to access the 3rd column of the 2nd row of the array *table*, use **table[1][2]** (since the first row and first column have index 0)

In the diagrams in **I.**, above, this is the element marked "x"

IV. The *length* Instance Variable

Every array object has an instance variable called *length* which stores the size (i.e., number of elements) of the array. Since a 2D array is a 1D array of 1D arrays, each array has its own length:

```
int table [][] = new int[4][5] ;        // 4 rows by 5 columns

int numRows = table.length ;            // numRows gets 4

int numCols = table[0].length ;         // numCols gets 5
```

In the above example, the length of the first dimension is 4 and the length of each array "pointed to" is 5.

V. Alternate Notation for 2D Array Declarations

As with 1D arrays, we may declare 2D arrays by specifying the initial values instead of the size

Java will infer the number of rows and columns from the initial values provided, as shown in this example:

```
// create a 3 by 4 array containing the ints 1 thru 12
int [][] table = {{1,2,3,4},        // 1st row (table[0])
                  {5,6,7,8},        // 2nd row (table[1])
                  {9,10,11,12}} ;   // 3rd row (table[2])
```

table:

0	1	2	3	4
1	5	6	7	8

2	9	10	11	12
	0	1	2	3

VI. Traversing a 2D Array

To *traverse*, or "visit" each element of a 2D array, nested *for* statements are commonly used.

To traverse by rows, the *outer* loop variable is used as the row index and the *inner* loop variable as the column index.

E.g. suppose array *matrix* has been declared as shown here

```
int [][] matrix = new int[4][3] ;          // 4 rows by 3 cols
```

The following code stores the ints 1 thru 12 in *matrix*, by rows, as shown below:

```
int count = 1 ;
// for each row...
for (int row = 0 ; row < matrix.length ; row++)
{
        // visit each column...
        for (int col = 0 ; col < matrix[0].length ; col++)
            matrix[row][col] = count++ ;
}
```

matrix:

0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12
	0	1	2

To traverse by columns, the *outer* loop variable is used as the column index and the *inner* loop variable as the row index.

```
    // store the ints 1 thru 12 in matrix, by columns
count = 1 ;
// for each column...
for (int col = 0 ; col < matrix[0].length ; col++)
{
        // visit each row...
        for (int row = 0 ; row < matrix.length ; row++)
            matrix[row][col] = count++ ;
}
```

matrix

0	1	5	9
1	2	6	10
2	3	7	11
3	4	8	12
	0	1	2

VII. Accessing An Entire Row of a 2D Array

Consider the 2D array table, as initialized in **V.**, above, and the following statements:

```
    // declare a 1D array object-variable pointing to the first
  // row of table
    int [] firstRow = table[0] ;
    // now create a deep copy of the first row of table
    int [] copy = Arrays.copyOf( table[0], table[0].length ) ;
```

The array object-variable *firstRow* contains a reference to the first row of table (i.e. it is a "pointer" to the array table[0]). So if these statements were executed:

```
    firstRow[1] = 37 ;
    System.out.println( table[0][1] ) ;
```

The output would be 37, because firstRow[1] and table[0][1] both refer to the same memory location.However, this statement:

System.out.println(copy[1]) ;

Will print the original value, 2, because the array *copy* is a *duplicate* of the array *table[0]* and was not modified.

BACKGROUND.

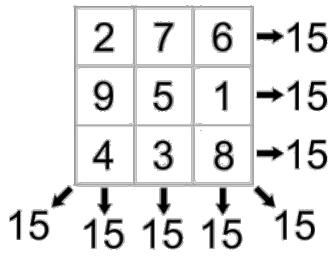
One interesting application of two-dimensional arrays is magic squares. A magic square is a square matrix in which the sum of every row, every column, and both diagonals is the same. Magic squares have been studied for many years, and there are some particularly famous magic squares. In this exercise, you will write code to determine whether a square is magic.

The folder **/src** contains the following files:

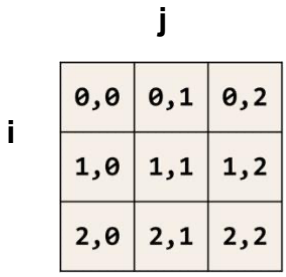
- **SquareTest.java:** This class tests the functionality of the class Square. It contains the shell for a program that reads input for squares from a file named **magicData** and tells whether each is a magic square. Note that the main method reads the size of a square, then after constructing the square of that size, it calls the *readSquare* method to read the square in. The *readSquare* method must be sent the Scanner object as a parameter.
- **Square.java:** Interface describing the ADT square, which has the following functionalities: 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. Note that the read method takes a Scanner object as a parameter.
- **My2DArraySquare.java:** This file contains the shell for the class that implements the square interface with a 2D array. It contains headers for a constructor that gives the size of the square and implement the functions of the interface. The read method is given for you; you will need to implement the other methods. You also have to specify the only attribute of the class: a 2D array.
- **magicData:** This file contains the squares. The first line of the file contains a number representing the size of the magic square that follows. Note that the -1 at the bottom tells the test program to stop reading.

You should find that the first, second, and third squares in the input are magic, and that the rest (fourth through seventh) are not. All the files can be found in the folder **/src**. Read more about magic square in https://en.wikipedia.org/wiki/Magic_square.

-Example of a magic square:



TIP: Remember that, in general, it is very useful to draw a little example of the data structure in order to help you to figure out how to solve the problem analysed. For example:



EXERCISE.

Implement the **SquareTest.java** and **My2DArraySquare.java** classes. Look for the comments "TO-DO" in the code and implement such part of the class yourself.

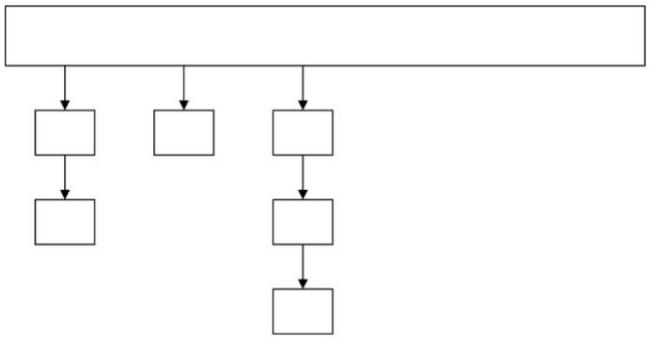
SUBMISSION.

Submit in Canvas **SquareTest.java** and **My2DArraySquare.java** before your lab on week 4.

Part 2: Learning to use an array of linked lists **(Week 3)**

PREVIOUS STUDY.

An array of linked lists is defined as a collection of nodes that can be traversed starting at the head node. It is important to note that head is not a node, rather the address of the first node of the list. Linked lists are very useful in situations where the program needs to manage memory very carefully and a contiguous block of memory is not needed. An array of linked lists is an important data structure that can be used in many applications. Conceptually, an array of linked lists looks as follows.



An array of linked list is an interesting structure as it combines a static structure (an array) and a dynamic structure (linked lists) to form a useful data structure. This type of structure is appropriate for applications, where for example, number of categories is known in advance, but how many nodes in each category is not known. For example, we can use an array (of size 26) of linked lists, where each list contains words starting with a specific letter in the alphabet.

Let's see a practical example of how to implement our dictionary. Remember that all objects in Java are stored as references. Therefore, you can create an array of LinkedList objects easily; you cannot, however, alter the number of elements the array can store once it has been created. (By default, all indices are set to null.) You can do this as follows:

```
LinkedList[] dict = new
LinkedList[NUM_ELEMENTS]; dict[0] =
linkedListObject1; dict[1] = linkedListObject2;
// ...
```

Creating an array of objects is no different than creating an array of primary data types. Remember:

```
int[] a = new int[7];  //create the array
a[0] = 0;  // intialize indexes
a[1] = 1;
...
```

As you should know, this creates an array of 7 integers. Creating an array of LinkedList objects is no different. Since you already know how to implement a LinkedList by yourself (from the module Linear Data Structures & Algorithms), for this module you are allowed to use the library LinkedList from Java and use all its methods (or you can use as well your implementation of LinkedList):

```
import java.util.LinkedList;
```

Remember that the LinkedList library implements several functions, such as “get”, “size”, etc:

```
dictionary[0].get(0);
int size = dictionary[0].size();
dictionary[1].add("ball");
```

TIP: You can use the Java Library LinkedList. Read the documentation of this library or any other that you might need in the future from the following website https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html. In Eclipse, press “.” after creating an object for seeing all the methods implemented.

```
public class LinkedList<E> {
      int size = 0;
      Node<E> first; // Pointer to first node.
      Node<E> last; // Pointer to last node.
      public LinkedList() {…}// Constructs an empty list.
      …
}
```

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;
    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

NOTE: You do not have to implement the functions that we studied for my_Linked_List you just have to use the functions from the library (or your implementation of Linked List from the previous module). Check the names of the available functions and their functionality and use only the ones that are helpful for solving the problem.

Following, we define a new array of linked lists called “dictionary”:

```
LinkedList <String> [] dictionary = new LinkedList[26]  //create the array
```

Now, you must remember all that this line does is to ceate the pointers in memory. None of these indexes are initialized. In order to actually use the linked lists, you will have to initialize each one:

```
for(int i = 0; i < dictionary.length; i++)
{
        dictionary[i] = new LinkedList<String> ();
}
```

Your dictionary is now ready to go! Now we just use it as we would any linked list:

```
dictionary[0].add("apple");
dictionary[0].add("amazing");
dictionary[1].add("ball");
...
```

BACKGROUND.

Solve the problem of the magic square by implementing the interface Square with an array of linked lists. The same methods of the previous lab (see lab01-02.pdf for more information) must be implemented.

The folder **/src** contains the following file:

- **MyArrayLinkedListsSquare.java:** This file contains the shell for the class that implements the square interface with an array of linked lists. It contains headers for a constructor that gives the size of the square and implement the functions of the interface. You also have to specify the only attribute of the class: an array of linked lists.

Reuse the following files:

- **SquareTest.java:** This class tests the functionality of the class Square. You must change the constructor of the objects Square, so that you are using the implementation MyArrayLinkedListsSquare.java.

  IMPORTANT: change the initial assignment of variable *type* to *false*.

- **Square.java:** Interface describing the ADT square.
- **magicData:** This file contains the squares. Note that the -1 at the bottom tells the test program to stop reading.

TIP: Remember that, in general, it is very useful to draw a little example of the data structure in order to help you to figure out how to solve the problem analysed.

Note that the results must be the same as with the previous implementation.

EXERCISE.

Implement the **MyArrayLinkedListsSquare.java**. Look for the comments "TO-DO" in the code and implement such part of the class yourself. Also, remember to change the constructor

of the objects Square in the file **SquareTest.java**, so that you are using the implementation **MyArrayLinkedListsSquare.java.**

FINAL QUESTION: After implementing these 2 different data structures for the ADT Square (from the families static implementation and dynamic implementation), which one would you had chosen if you were told to solve the magic square with total freedom to use any type of implementation? Why?

SUBMISSION.

Submit in Canvas **MyArrayLinkedListsSquare.java** and a document with your answer for the final question before your lab on week 4.