CPSC 150L Lab 8
2D Arrays

Fall 2017

1 Introduction

When designing a program, we need to understand how to handle massive quantities of data. The most fundamental way to handle large quantities of data is to use an *array.* An array is a structure which most programming languages have, and thus is an important tool to study and understand.

2 Exercises

Fork and clone the `cpsc150l-lab08` project from the 150 students group on Gitlab.

2.1 Two Dimensional Arrays

Create a class called `Array2DMethods.java` and implement the following methods.

1. `public static int[] sumRows(int[][] a)`

   Returns an array containing the sum of each row. Furthermore, the index of each sum in the returned array must correspond to its row's index. That is, the sum of row i

must be the ith element of the resultant array. Moreover, if we receive `null` return `null`. If you receive an empty array, return `new int[0]` (this returns an empty array). If a row is empty, then its sum is `0`. For example if the matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

was the input, we should get an output of `{6,15,24}`.

2. `public static int[] getRow(int[][] a, int n)`

   Returns the array at row `n` if `n` is within the boundaries of `a`. Otherwise returns `null`. For example if

$$a = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

   then we'd get the following table as output.

n	0	1	5
return	{1,2,3}	{4,5,6}	null

3. `public static double average(int[][] a)`

   Returns the average of all elements in the array `a`. If we receive `null` or the empty array, return `0`.

Exercise 1 Complete

Run:

```
git add .
git commit -m "Completed exercise 1"
git push origin master
```

3 Common Mistakes

The common mistakes for this lab are as follows.

1. Remember that the first element of an array in Java has an index of `0`.
2. When accessing elements of an array ensure your index is between `0` and `array.length - 1` inclusive.
3. You can check whether or not an integer is odd or even by computing it modulo 2.
4. 2D arrays are indexed as `array[row][column]`.
5. If a method fails to execute, it is likely a null pointer exception or an array out of bounds exception.

4 Tutorial

4.1 Arrays

In programming, an *array* is an **ordered list** of data. In Java, to declare an array, `x`, of a given type `T` with length `n` we use the following.

Code:

```
T[] x = new T[n];
```

For example, we can declare an integer array of length 6 by executing

Code:

```
int[] array = new int[6];
```

which allocates a contiguous block of memory that looks like the following.

0	0	0	0	0	0

When you initialize an array like this, notice that it is similar to how you initialized a `Scanner` in previous labs. When you initialize an array or object, Java allocates a block of memory to it and places a *reference* to that block in the variable. In the case of an array, that reference stores the address of the first item in the array. In order to access that item you need to give it an *offset*. In Java, among many other languages, in order to access the first element of the array, you give it an offset of 0 like so.

Code:

```
array[0]
```

If you wish to access the `k`th element, you pass it an offset of `k-1`.

Code:

```
array[k-1]
```

When you declare an array of primitive types (`int`, `char`, `double`, `boolean`, etc.) the compiler automatically fills the array with 0, or in the case of `boolean`, `false`. If we consider this code again,

Code:

```
int[] array = new int[6];
```

we will see the following in memory.

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]
0	0	0	0	0	0

Another way to declare an array is to use

Code:

```
int[] array = {0,1,2,3,4,5};
```

which yields the following.

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]
0	1	2	3	4	5

When you access elements of an array, you need to ensure that you do not go out of bounds. For example, the following code will give you an out of bounds error.

Code:

```
int[] arr = new array[10];
arr[10] = 0;
```

In Java, you do not need to manually track the size of any array as the JVM stores it. You can get the size of an array, `x`, by using `x.length`. Note that this is the **number of elements in the array**, therefore the largest legal offset is `x.length - 1`. Once you access the element you want, you can do whatever you wish to it whether it is writing to that location or reading from that location. Consider the following example.

Code:

```
int[] arr = new int[10]
for (int i = 0 ; i < arr.length ; i++)
     arr[i] = 2 * (i+1);
int sum = 0;
for (int i = 0 ; i < arr.length ; i++)
     sum = sum + arr[i];
```

In the first loop, we *initialize* the array `arr`. After that loop executes, we will have the following in memory.

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]	arr[6]	arr[7]	arr[8]	arr[9]
2	4	6	8	10	12	14	16	18	20

After the second loop executes, `sum` will have the value `110`.

4.2 Arrays with Loops

You saw how count-controlled loops can play nicely with arrays, now we will use a `while` loop to find the index of the first `0` in the array.

Code:

```
public static int findZero(int[] arr) {
     int curElement = 0;
     while (curElement < arr.length && arr[curElement] != 0)
          curElement++;
     if (curElement == arr.length)
          return -1;
     else
          return curElement;
}
```

Since we are using a while loop, we needed to add the `curElement < arr.length` to the condition in order to ensure that we stayed within the bounds of the array. Furthermore, we also had to increment our counter inside the loop.

Additionally, we have another type of loop that is specially designed for arrays, the *for each loop*. This loop operates *for each* element in the array. Given an array of type `T` named `arr`, we would use a for each loop as follows.

Code:

```
for (T elt : arr) {
    // Do operation
}
```

This type of loop will **never** go out of bounds, and thus is safe to use. However, for each loops are unaware of the index of an element and therefore cannot be used to compare nearby elements. Furthermore for each loops are *read only*, meaning you cannot alter an array using a for each loop. In spite of these drawbacks, they are very useful tools. For example, to sum the elements of an array with a for each loop is as follows.

Code:

```
public static int sum(int[] arr) {
     int sum = 0;
     for (int i : arr)
          sum += i;
     return sum;
}
```

Note how we do not index anything in the array, therefore we cannot get an out of bounds error. If we were to pass this method an empty array, it would also return a value of 0, which is what we want.

Lastly, consider the following code. Recall that `a % 2` is 0 if `a` is even and 1 if `a` is odd.

Code:

```
public static void rightShift(String[] arr) {
     String[] tmp = {"",""};
     for (int i = 0 ; i < arr.length ; i++){
          tmp[i % 2] = arr[i];
          if (i == 0)
               arr[0] = arr[arr.length - 1];
          else
               arr[i] = tmp[(i+1) % 2];
     }
}
```

This method has a return type of void, and thus does not return anything. However, after executing this method on the array

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]
"0"	"1"	"2"	"3"	"4"	"5"

we will see the following in memory.

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]
"5"	"0"	"1"	"2"	"3"	"4"

This is because we modified the memory in the address which the reference `arr` holds. Specifically, this method takes an array and shifts all elements one index to the right. In this method, we use *temporary variables* in order to store array elements that get overwritten by the right shift. In order to swap two elements in an array we perform the following.

Code:

```
public static void swapValues(int[] arr, int i, int j) {
     if (0 <= i && i < arr.length && 0 <= j && j < arr.length) {
          int temp = arr[i];
          arr[i] = arr[j];
          arr[j] = arr[i];
     }
}
```

We store the first element in a temporary variable, then overwrite the first element with the second element, and lastly overwrite the second element with the temporary variable.

4.3 Two Dimensional Arrays

So far, the only arrays we have dealt with have been *one dimensional.* However, we can have two dimensional arrays. We do this by declaring an array of arrays, which in Java looks as follows.

Code:

```
T[][] arrayName = new T[n][m];
```

The previous code will create `n` arrays of length `m`, or in other words it creates a table with `n` rows and `m` columns. If we see

Code:

```
arrayOfArrays[n]
```

it accesses the `n`th array in the table. The code

Code:

```
arrayOfArrays[n][m]
```

accesses the `m`th element of the `n`th array in the table. Consider the following code.

Code:

```
public static int matrixSum(int[][] matrix) {
     int sum = 0;
     for (int row = 0 ; row < matrix.length ; row++) {
          for (int column = 0 ; column < matrix[0].length ; column++) {
               sum += matrix[row][column];
          }
     }
     return sum;
}
```

In the previous method, we sum all the elements of an integer matrix. In order to do so, we sum the elements of each array in the matrix. We can also do this with for each loops in a similar fashion.

Code:

```
public static int matrixSum(int[][] matrix) {
     int sum = 0;
     for (int[] row : matrix) {
          for (int element : row) {
               sum += element;
          }
     }
     return sum;
}
```

Additionally, note that all pitfalls that apply to one dimensional arrays apply to two dimensional arrays.