

2023-02-13

Warm up

Sample test questions

_____ 1) Consider the following code segment.

```
int[][] arr = {{1, 2, 3, 4},
               {1, 2, 3, 4}};
for (int j = 0; j < arr.length; j++)
{
    for (int k = 0; k < arr[0].length; k++)
    {
        if (k > 0 && arr[j][k] > 10 / k)
        {
            System.out.print("*");
        }
    }
}
```

How many times will "*" be printed when the code segment is executed?

- (A) 1
- (B) 2
- (C) 3
- (D) 4
- (E) It will not be displayed because there is a divide by 0 exception.

_____ 2) Consider the following code segment, where letters is a two-dimensional (2D) array that contains possible letters. The code segment is intended to print "FIB".

```
String[][] letters =      {{"A", "B", "C"},  
                           {"D", "E", "F"},  
                           {"G", "H", "I"}};
```

```
System.out.println( /* missing code */ );
```

Which of the following could replace `/* missing code */` so that the code segment works as intended?

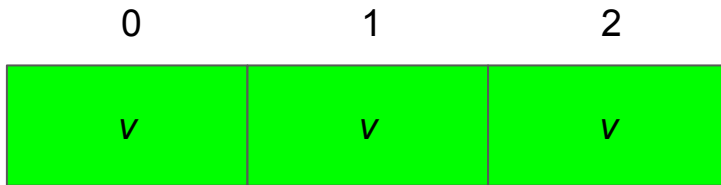
- (A) `letters[2][3] + letters[2][2] + letters[0][1]`
- (B) `letters[1][2] + letters[3][3] + letters[1][2]`
- (C) `letters[0][1] + letters[1][1] + letters[0][1]`
- (D) `letters[1][2] + letters[2][2] + letters[0][1]`
- (E) `letters[2][3] + letters[3][3] + letters[1][2]`

8.1

Two-Dimensional Arrays

Arrays

Arrays are a zero-based indexed sequence of values of the same type. Any Java type! (Well, not void. But all other primitive or reference types.)



```
type[] name;
```

Examples:

```
boolean[] answers;  
String[] questions;  
int[] scores;  
Student[] students;
```

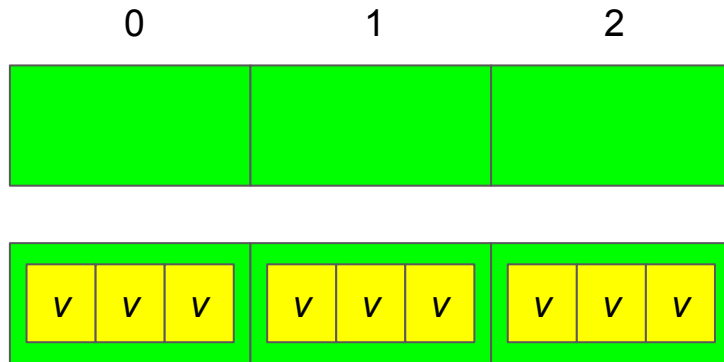
Array of Arrays

Arrays are a type, too. So, an array can be declared with arrays as its values. The result? Two-dimensional arrays!

```
type[][] name;
```

Examples:

```
boolean[][] theaterSeats;  
String[][] seatingChart;  
int[][] bingoCard;  
Apt[][] building;
```



2D Arrays

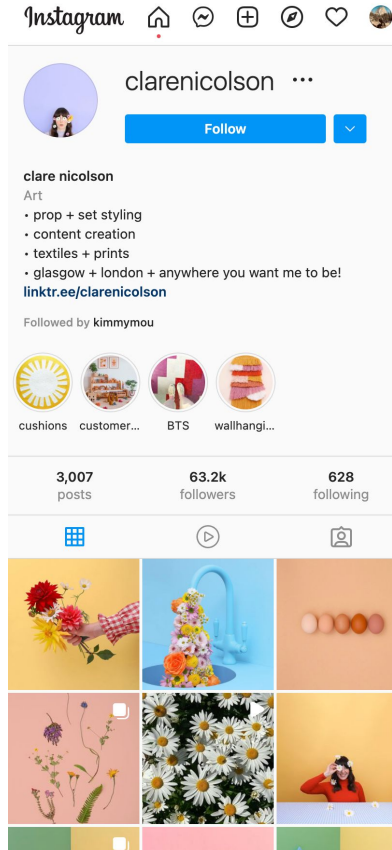
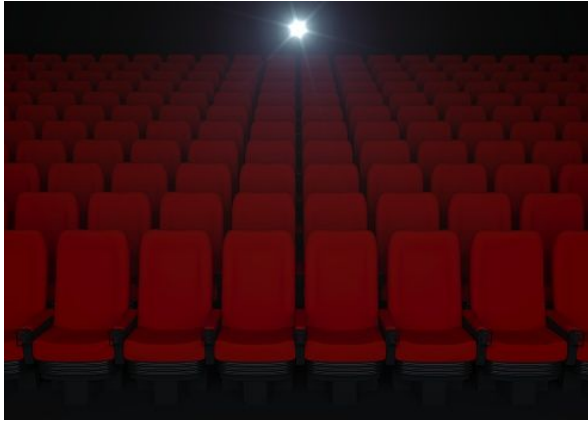
In Java, a two-dimensional array is really a one-dimensional array of one-dimensional arrays.

Conceptually, though, such arrays are used to model two-dimensional concepts, and we think of it as a single table with rows and columns.

At the Java level, it's actually an outer array where each element is a reference to a nested inner array.



What are 2D arrays good for?



Array Storage

Many programming languages actually store two-dimensional array data in a one-dimensional array.

- row-major order
- column-major order

In C/C++, for instance, you can declare

```
int a[3][3];
```

and under the hood, the C compiler will make a one-dimensional array A' that is size 9, and compile $a[i][j]$ to $A'[i*3 + j]$

The frame buffer that stores the pixels on your screen is obviously 2D, but is probably stored in a one-dimensional array referenced by $\text{frameBuffer}[y * \text{screenWidth} + x]$

The Java language designers made a decision to model multi-dimensional arrays as arrays of arrays. (Python: Same.)

| | 0 | 1 | 2 |
|---|---|----|----|
| 0 | 2 | 4 | 6 |
| 1 | 8 | 10 | 12 |

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|----|----|
| 2 | 4 | 6 | 8 | 10 | 12 |

Row-Major Order

| | | | | | |
|---|---|---|----|---|----|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 2 | 8 | 4 | 10 | 6 | 12 |

Column-Major Order

For the ASCII Art exercise, it is possible to implement the Canvas using a one-dimensional array in row or column major order.

In some respects, this is more efficient, as it's one contiguous object in memory.

Accessing the elements is more complicated, so the accessor/mutator methods become more important!

```
class Canvas {
    private String name;
    private char pixels[];
    private int width, height;

    public Canvas(String name, int width, int height) {
        this.name = name;
        this.width = width;
        this.height = height;
        pixels = new char[width * height];
    }

    public String getName() { return name; }
    public int getWidth() { return width; }
    public int getHeight() { return height; }

    public void setPixel(int rowIdx, int colIdx, char pixel) {
        pixels[width * rowIdx + colIdx] = pixel;
    }

    public Character getPixel(int rowIdx, int colIdx) {
        return pixels[width * rowIdx + colIdx];
    }
}
```

Here's the more conventional
2D array approach to the
ASCII Art Canvas.

Is pixels in row-major or
column-major order here?

```
class Canvas {  
    private String name;  
    private char pixels[][];  
    private int width, height;  
  
    public Canvas(String name, int width, int height) {  
        this.name = name;  
        this.width = width;  
        this.height = height;  
        pixels = new char[height][width];  
    }  
  
    public String getName() { return name; }  
    public int getWidth() { return width; }  
    public int getHeight() { return height; }  
  
    public void setPixel(int rowIdx, int colIdx, char pixel) {  
        pixels[rowIdx][colIdx] = pixel;  
    }  
  
    public Character getPixel(int rowIdx, int colIdx) {  
        return pixels[rowIdx][colIdx];  
    }  
}
```

Array types of array types

Every type `T` in Java has a related array type `T[]`

Examples: `int` and `int[]`, `String` and `String[]`

But... this is ALSO true for array types themselves!

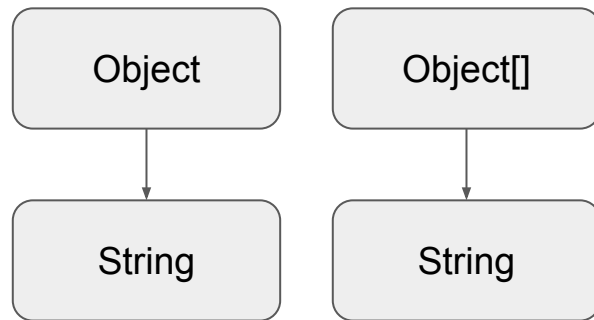
`T[]` has a related array type `T[][]`

`T[][]` is an array of `T[]`

```
type[][] name;
```

- Examples

```
boolean[][] theaterSeats; <- array of boolean[]  
String[][] seatingChart; <- array of String[]  
int[][] bingoCard; <- array of int[]  
Apt[][] building; <- array of Apt[]
```



Multi-Dimensional Arrays

You don't have to stop at two dimensions...

The type `T[[]]` is an array of `T[]`

Example: A 3D printer prints "voxels", three-dimensional pixels
Software for a 3D printer might store a model in memory in a 3-dimensional array.

```
Voxel[][[]] voxels;
```

A Minecraft-like game might store an animation as frames of 3D voxels, but over time. So, a 4-dimensional array:

```
Voxel[][[]][[]] voxelAnimationFrames;
```

Two-dimensional arrays are the most common multi-dimensional arrays, though.



Array of Arrays - Definition

```
boolean[][] theaterSeats = new boolean[numRows][numSeatsPerRow];
```

```
String[][] seatingChart = new String[numRows][numSeatsPerRow];
```

```
int[][] bingoCard = new int[5][5];
```

```
Apt[][] building = new Apt[numFloors][numAptsPerFloor];
```

*** NOTE 1 ***

*Two-Dimensional
Arrays can have
non-equal
dimensions!*

```
boolean[][] theaterSeats = new boolean[75][25];
```

```
String[][] seatingChart = new String[5][10];
```

```
int[][] multiplicationTable = new int[100][500];
```

```
Apt[][] building = new Apt[5][10];
```

Array of Arrays - Definition

```
boolean[][] theaterSeats = new boolean[numRows][numSeatsPerRow];
```

```
String[][] seatingChart = new String[numRows][numSeatsPerRow];
```

```
int[][] bingoCard = new int[5][5];
```

```
Apt[][] building = new Apt[numFloors][numAptsPerFloor];
```

*** NOTE 2 ***

*Just like Arrays -
Two-Dimensional
Arrays initialize
their values to
"reasonable"
defaults*

- 0 for numeric types
- null for Object types
- false for boolean types

Array of Arrays - Definition

```
int[][] bingoCard = new int[5][5];
```

Alternatively...

```
int[][] bingoCard = new int[5][]; // omit internal array size
bingoCard[0] = new int[5];
bingoCard[1] = new int[5];
bingoCard[2] = new int[5];
bingoCard[3] = new int[5];
bingoCard[4] = new int[5];
```

Array of Arrays - Definition

```
boolean[][] theaterSeats = new boolean[numRows][numSeatsPerRow];
```

Alternatively...

```
boolean[][] theaterSeats = new boolean[numRows][];  
for (int rowIdx = 0 ; rowIdx < numRows ; rowIdx++ {  
    theaterSeats[rowIdx] = new boolean[numSeatsPerRow];  
}
```

Arrays of Arrays - Example

```
boolean[][] theaterSeats = new boolean[rows][seats];
```

`theaterSeats[0]`

| <code>theaterSeats[0][0]</code> | <code>theaterSeats[0][...]</code> | <code>theaterSeats[0][seats-1]</code> |
|---------------------------------|-----------------------------------|---------------------------------------|
| false | false | false |

`theaterSeats[...]`

| <code>theaterSeats[...][0]</code> | <code>theaterSeats[...][...]</code> | <code>theaterSeats[...][seats-1]</code> |
|-----------------------------------|-------------------------------------|---|
| false | false | false |

`theaterSeats[rows-1]`

| <code>theaterSeats[rows-1][0]</code> | <code>theaterSeats[rows-1][...]</code> | <code>theaterSeats[rows-1][seats-1]</code> |
|--------------------------------------|--|--|
| false | false | false |

Write: `theaterSeats[0][0] = true;`

Read : `System.out.println(theaterSeats[rows-1][seats-1]);`

Arrays of Arrays - Initializer Lists

- You can initialize the values of a Two-Dimensional Array when you create it (and the sizes will be automatically calculated)

```
int[][] ticketInfo = { {25,20,25}, {25,20,25} };  
ticketInfo.length      => 2  
ticketInfo[0].length) => 3  
ticketInfo[1].length) => 3
```

```
boolean[][] jaggedTable = { {false, true, true}, {false}, {true} };  
jaggedTable.length      => 3  
jaggedTable[0].length) => 3  
jaggedTable[1].length) => 1  
jaggedTable[2].length) => 1
```

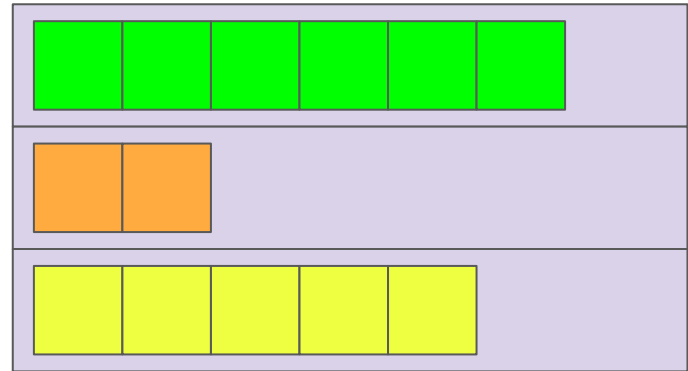
Arrays of Arrays - Jagged Arrays

- On the exam all inner arrays will have the same length - **However** it is possible in Java to have inner arrays of different lengths. These are called Jagged (or Ragged) Arrays

```
int jaggedTable[][] = new int[3][]; // omit the inner array size
```

```
jaggedTable[0] = new int[6];  
jaggedTable[1] = new int[2];  
jaggedTable[2] = new int[5];
```

```
jaggedTable.length      => 3  
jaggedTable[0].length   => 6  
jaggedTable[1].length   => 2  
jaggedTable[2].length   => 5
```



8.2

Traversing Two-Dimensional Arrays

Review: Traversing Arrays with `for` loops

- Remember that the range of valid Array indexes (for non-empty Arrays) is 0 to `Array.length - 1`

```
int scores[] = {95, 100, 91, 85 };  
for (int idx = 0; idx < scores.length; idx++)  
{  
    System.out.println(scores[idx]);  
}
```



```
int scores[] = {95, 100, 91, 85 };  
for (int idx = 1; idx <= scores.length;  
idx++) {  
    System.out.println(scores[idx]);  
}
```



Review: Traversing Arrays with `for` loops

- Remember that the range of valid Array indexes (for non-empty Arrays) is 0 to `Array.length - 1`

```
int scores[] = {95, 100, 91, 85 };  
for (int idx = 0; idx < scores.length; idx++)  
{  
    System.out.println(scores[idx]);  
}
```



```
int scores[] = {95, 100, 91, 85 };  
for (int idx = 1; idx <= scores.length;  
idx++) {  
    System.out.println(scores[idx]);  
}
```



Note: Passing an out of range index will cause a `ArrayIndexOutOfBoundsException`!

Review: Traversing Arrays with `for` loops

- Remember that the range of valid Array indexes (for non-empty Arrays) is 0 to `Array.length - 1`

```
int scores[] = {95, 100, 91, 85 };  
for (int idx = 0; idx < scores.length; idx++)  
{  
    System.out.println(scores[idx]);  
}
```

```
int scores[] {95, 100, 91, 85 };  
for (int idx = 1; idx <= scores.length;  
idx++) {  
    System.out.println(scores[idx]);  
}
```



***This loop also
skips the first
element in the
Array!***

Note: Passing an out of range index will cause a `ArrayIndexOutOfBoundsException`!

Review: Traversing Arrays with `for` loops

- You can use a `for` loop to traverse an Array from back to front!

```
int scores[] = {95, 100, 91, 85 };  
for (int idx = scores.length - 1; idx >= 0; idx--) {  
    System.out.println(scores[idx]);  
}
```

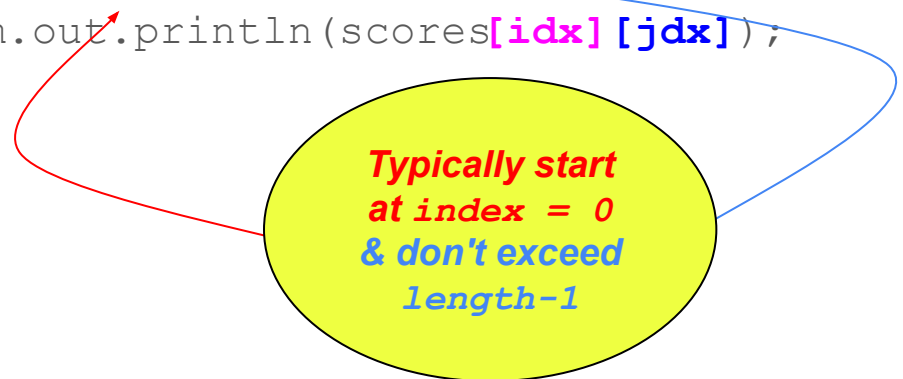
- ...or to traverse any arbitrary range of elements

```
int scores[] = {95, 100, 91, 85 };  
for (int idx = 1; idx <= 2; idx++) {  
    System.out.println(scores[idx]);  
}
```

Traversing Two-Dimensional Arrays with `for` loops

- Traversing Two-Dimensional Arrays is very similar

```
int scores[][] = {{10,20,30},{40,50,60}};  
for (int idx = 0; idx < scores.length; idx++) {  
    for (int jdx = 0; jdx < scores[idx].length; jdx++) {  
        System.out.println(scores[idx][jdx]);  
    }  
}
```



Typically start
at index = 0
& don't exceed
length-1

Note: Passing an out of range index will cause a `ArrayIndexOutOfBoundsException`!

Review: Traversing Arrays with for-each loops

```
for (type arrayItemVariable : arrayVariable) {  
    arrayItemVariable resolves to arrayVariable[...]  
}
```

```
String[] colors = {"red", "orange", "purple"};
```

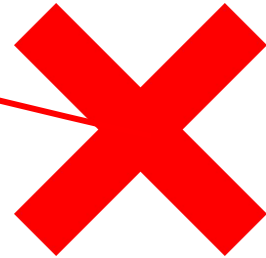
```
System.out.println("begin");  
for (String color: colors) {  
    System.out.println(" " + color);  
}  
System.out.println("end");
```

Review: Traversing Arrays with for-each loops

- The type of the for-each variable MUST match the type of the values stored in the Array

```
String colors[] = {"red", "orange", "purple"};
```

```
for(int color: colors) {  
    System.out.println(" " + color);  
}
```



Note: color must be of type String since colors is an Array that contains Strings

Traversing Two-Dimensional Arrays with for-each loops

- Remember during the introduction of Two-Dimensional Arrays - We said:
"Arrays are a type - Which means you can easily create an Array that contains Arrays - often called Two-Dimensional Arrays"
- That means we can use `for-each` to traverse a Two-Dimensional Array almost exactly like a One-Dimensional Array (**we just have to be careful how we declare the types**)

Reminder: `for-each` loops can be super-useful; but you are unable to make use of an index or change the underlying Array while looping

Traversing Two-Dimensional Arrays with for-each loops

Given this Two-Dimensional Array

```
int scores[][] = {{10,20,30},{40,50,60}};
```

And this general description of for-each

```
for (type arrayItemVariable : arrayVariable) {  
    arrayItemVariable resolves to Array[...]  
}
```

Q: What is the type of the outer array in scores?

scores[] -> an array of int[]

Q: What is the type of the inner array in scores?

scores[][] -> an array of int

Traversing Two-Dimensional Arrays with for-each loops

```
int scores[][] = {{10,20,30},{40,50,60}};
```

`scores[]` -> an array of `int[]`

`scores[][]` -> an array of `int`

So we can use for-each to traverse the Two-Dimensional Array like this

```
for (int[] outer : scores) {  
    for (int inner : outer) {  
        System.out.println(inner);  
    }  
}
```

**Note: The inner array is a One-Dimensional Array -
So we use the same for-each that we used
previously for One-Dimensional Arrays**

No indices available for use!

```
for (type arrayItemVariable : arrayVariable) {  
    arrayItemVariable resolves to arrayVariable[...]  
}
```


Traversal of Jagged Arrays

- Traversal of Two-Dimensional Jagged Arrays works the same!

```
int jaggedTable[][] = new int[3][];
```

```
jaggedTable[0] = new int[]{4,1,5,9,6,3};
```

```
jaggedTable[1] = new int[]{9,1};
```

```
jaggedTable[2] = new int[]{0,3,2,9,4};
```

```
for (int idx = 0; idx < jaggedTable.length; idx++) {  
    for (int jdx = 0; jdx < jaggedTable[idx].length; jdx++) {  
        System.out.print(jaggedTable[idx][jdx] + " ");  
    }  
    System.out.println();  
}
```

| | | | | | |
|---|---|---|---|---|---|
| 4 | 1 | 5 | 9 | 6 | 3 |
| 9 | 1 | | | | |
| 0 | 3 | 2 | 9 | 4 | |

Example: Basketball Scores

```
int NUM_PLAYERS = 5, NUM_GAMES = 3;  
int scores[][] = new int[NUM_PLAYERS][NUM_GAMES];
```

| | Game 1 | Game 2 | Game 3 |
|----------|--------|--------|--------|
| Player 1 | 14 | 19 | 22 |
| Player 2 | 24 | 13 | 5 |
| Player 3 | 5 | 26 | 31 |
| Player 4 | 0 | 18 | 40 |
| Player 5 | 15 | 9 | 46 |

`scores[][]`

Example: Basketball Scores

```
int NUM_PLAYERS = 5, NUM_GAMES = 3;  
int scores[][] = new int[NUM_PLAYERS][NUM_GAMES];
```

| | Game 1 | Game 2 | Game 3 |
|----------|--------|--------|--------|
| Player 1 | 14 | 19 | 22 |
| Player 2 | 24 | 13 | 5 |
| Player 3 | 5 | 26 | 31 |
| Player 4 | 0 | 18 | 40 |
| Player 5 | 15 | 9 | 46 |

```
scores[3][0], scores[3][1], scores[3][2]
```

Example: Basketball Scores

```
int NUM_PLAYERS = 5, NUM_GAMES = 3;  
int scores[][] = new int[NUM_PLAYERS][NUM_GAMES];
```

| | Game 1 | Game 2 | Game 3 |
|----------|--------|--------|--------|
| Player 1 | 14 | 19 | 22 |
| Player 2 | 24 | 13 | 5 |
| Player 3 | 5 | 26 | 31 |
| Player 4 | 0 | 18 | 40 |
| Player 5 | 15 | 9 | 46 |

```
scores[0][1], scores[1][1], scores[2][1], scores[3][1], scores[4][1]
```

Example: Basketball Scores

```
int NUM_PLAYERS = 5, NUM_GAMES = 3;  
int scores[][] = new int[NUM_PLAYERS][NUM_GAMES];
```

| | Game 1 | Game 2 | Game 3 |
|----------|--------|--------|--------|
| Player 1 | 14 | 19 | 22 |
| Player 2 | 24 | 13 | 5 |
| Player 3 | 5 | 26 | 31 |
| Player 4 | 0 | 18 | 40 |
| Player 5 | 15 | 9 | 46 |

Q1: How would you determine the total points scored by all 5 players in all 3 games?

Q2: How would you determine which player had the highest number of points in Game 3?

Q3: How would you determine the average points scored by Player 3 in all 3 games?

InsertionSort exercise with 2D arrays

In the InsertionSort exercise, we used a Record class to represent each row of CSV data.

An alternative representation would be a 2D array: the outer array is the rows of data, and each row is an inner array with the columns.

Why? Sometimes you don't need to be able to look up columns easily by name, such as if you're just formatting it.

```
private String[][] loadRecords() throws IOException {
    ArrayList<String[]> records = new ArrayList<String[]>();
    FileInputStream fileInputStream = new FileInputStream(new File("fortune500.csv"));
    Scanner scanner = new Scanner(fileInputStream);
    String header = scanner.nextLine(); // Skip the header
    while (scanner.hasNextLine()) {
        String line = scanner.nextLine();
        if (line.isEmpty()) {
            continue;
        }
        String[] row = line.split(",");
        records.add(row);
    }
    String[][] result = new String[records.size()][];
    records.toArray(result);
    return result;
}
```

Could this result in a jagged array?

InsertionSort exercise with 2D arrays

The inner arrays in a multi-dimensional array in Java are just objects. You can do anything with them that you'd do with any object in an array, including reassign them and move them around.

Here, we're insertion sorting a 2D array, by rearranging the inner arrays.

```
// Precondition: "records" is a fully-populated array of String[] objects, from records[0]
// to records[records.length-1]
private void insertionSort(String[][] records,
                           int sortColumn) {
    for (int i=1, n=records.length; i<n; i++) {
        String[] temp = records[i];
        int j = i;
        while (j > 0 && records[j-1][sortColumn].compareTo(temp[sortColumn]) > 0) {
            records[j] = records[j-1];
            j--;
        }
        records[j] = temp;
    }
}
```

InsertionSort exercise with 2D arrays

Here, we're outputting the sorted results to a file.

The for-each iteration over the outer array came in handy here, as well as for the inner array.

This could've been expressed with regular for loops with i and j variables, but this is nice and clean.

```
private void run() throws IOException {
    String[][] records = loadRecords();
    insertionSort(records, 4);
    FileOutputStream fileOutputStream = new FileOutputStream(new File("output.txt"));
    PrintWriter pw = new PrintWriter(fileOutputStream);
    pw.println("ranking,lon,lat,company,industry,state,city");
    for (String[] record : records) {
        boolean first = true;
        for (String column : record) {
            if (!first) {
                pw.print(',');
            }
            first = false;
            pw.print(column);
        }
        pw.println();
    }
    pw.close();
}
```


Back to Battleship – Hints from Chris, one update

- #2: Using `Player.getDamageReport()` is not going to get you what you need; It does not display ship positions
- #4: For folks writing ship placement code - If you want to save some time check out `Ship.placeShip()` and `Grid.isLegal()`
- #5-6: Your AI - `Player.selectTarget()` - **SHOULD NOT** be accessing `Cell.getShip()` - **restrict your usage to** `Cell.isFiredOn()`, `Cell.isHit()`, and `Cell.isMiss()`
- UPDATE from Gary: It is OK to call `Cell.getShip()`, since you are told when you get a hit which ship it is, e.g. Tom: "D-4." Frank: "Hit. Destroyer." But don't look at the `locations` array in `Ship` because that would be cheating! The AI can know which ship was hit, but not where it is other than where it's already landed hits.

Battleship - One more targeting idea

- The instructions for the project outline a "Basic" and "Advanced" algorithm. The Basic algorithm is fine to implement.
- A middle ground:
 - Find a cell that has a hit and the ship isn't sunk yet.
 - If there is an adjacent cell with a hit on the same ship, and the opposite direction hasn't been fired on yet, fire in that opposite direction.
 - Otherwise fire on any adjacent cell. If none left, resume loop.
 - This will still fire at more spots than strictly necessary but will sink ships faster than the "Basic" algorithm.

For the rest of the period...

Continue work on Unit 8 Project

CS Awesome, study for Unit 8 Test