

Unit 8: 2D Array

Unit 8 covers two-dimensional arrays: how to declare them, how rows and columns are indexed, and how to traverse them with nested loops in row-major and column-major order. It has only two topics, which makes it the shortest unit, but it feeds directly into FRQ 4 and some of the trickiest tracing questions on the multiple-choice section.

AP Computer Science A

2D Array

About 12 minutes to read

How to use this guide

Read it in order the first time because the topics build on each other. Declaration and indexing come first, then traversal, then the small set of algorithms that traversal makes possible. Exam questions usually hand you a grid and a pair of nested loops and ask what the code prints or returns.

After the first read, use the trap boxes and the confusion table to review the distinctions the exam tests most often. Trace every code sample by hand with a pencil before you read the trace that follows it. Finish with the practice questions, then complete the recall check on the last page out loud and note any items you cannot explain yet.

What this unit is worth. Unit 8 is about 7.5 to 10 percent of the AP CSA exam. It is the smallest unit, but 2D array traversal is FRQ 4 every year (9 points), and nested-loop tracing on grids is a reliable MCQ trap. The row/column swap is the mistake that costs the most points for the least reason.

8.1 2D Arrays

A **2D array** is an array whose elements are themselves arrays. Picture a table with rows running across and columns running down, or a grid like a spreadsheet. Each cell holds one value and is named by two indices: which row it is in and which column it is in. Java stores it as an array of row arrays, which is why the row index always comes first.

To declare one, you write the type with two pairs of brackets and give both dimensions:

```
int[][] arr = new int[3][4];
```

This creates 3 **rows** and 4 **columns**, which is 12 ints, all initialized to 0. The first number is always rows. The second is always columns. Say it to yourself when you read a declaration: "3 rows, 4 columns."

Indexing is zero-based in both dimensions. `arr[0][0]` is the top-left cell. `arr[2][3]` is the bottom-right cell. Read `arr[r][c]` as "row r, column c." The r selects which row array, and then the c selects the position inside that row.

EXPRESSION	VALUE FOR NEW INT[3][4]	MEANING
<code>arr.length</code>	3	Number of rows
<code>arr[0].length</code>	4	Number of columns in row 0
<code>arr[2][3]</code>	0	Last cell, row 2 column 3
<code>arr.length * arr[0].length</code>	12	Total elements, rectangular grids only

Trap. `arr.length` is the number of rows, not the number of elements. For `new int[3][4]`, `arr.length` is 3 and the grid holds 12 values. Students who treat `arr.length` as "the size of the whole thing" write inner loops that stop early or trace the wrong number of iterations.

8.1 2D Arrays, continued

You can also build a 2D array with an **initializer list**, which fills in the values at declaration time. Each inner pair of braces is one row:

```
int[][] grid = { {1, 2, 3}, {4, 5, 6} };
```

This grid has 2 rows and 3 columns. Row 0 is {1, 2, 3} and row 1 is {4, 5, 6}, so `grid[1][2]` is 6. On the exam the braces are often written on one line, like `{{1, 2, 3}, {4, 5, 6}}`, which is the same array. Count the inner groups to find the number of rows, and count inside one group to find the number of columns.

Java technically allows each row to have a different length, which is called a **ragged array**:

```
int[][] ragged = { {1, 2}, {3, 4, 5}, {6} };
```

Here `ragged[0].length` is 2, `ragged[1].length` is 3, and `ragged[2].length` is 1. This is exactly why `arr[r].length` exists as a separate expression from `arr.length`: each row knows its own length. The AP exam uses rectangular grids, but correct code still uses `arr[r].length` as the inner loop bound, because that is the form that works for every grid.

Trap. The initializer needs one set of braces per row. `int[][] g = {1, 2, 3};` does not compile, because Java sees three ints where it expects three row arrays. When an initializer list looks wrong, check that every row is wrapped in its own braces.

8.2 Traversing 2D Arrays

Traversing a 2D array means visiting every element, and the standard way is a pair of **nested loops**: an outer loop over rows and an inner loop over columns.

```
for (int r = 0; r < arr.length; r++) {
    for (int c = 0; c < arr[r].length; c++) {
        // visit arr[r][c]
    }
}
```

The outer loop picks a row and holds it still while the inner loop walks across every column of that row. Then the outer loop advances to the next row. This visits cells left to right, top to bottom, which is called **row-major order**. Trace it on this grid:

```
int[][] m = { {2, 4}, {6, 8}, {10, 12} };
for (int r = 0; r < m.length; r++) {
    for (int c = 0; c < m[r].length; c++) {
        System.out.print(m[r][c] + " ");
    }
    System.out.println();
}
```

Row 0 prints 2 then 4, then a newline. Row 1 prints 6 then 8, then a newline. Row 2 prints 10 then 12, then a newline. The output is:

```
2 4
6 8
10 12
```

Trap. Writing the inner bound as `c < arr.length` is the quiet killer. On a 3-by-4 grid the inner loop would stop after 3 columns and silently skip the last column of every row. No error, no exception, just wrong answers. The inner loop runs over columns, so its bound is `arr[r].length`. The exam loves non-square grids precisely because they expose this mistake.

8.2 Traversing 2D Arrays, continued

To visit the grid in **column-major order**, top to bottom then left to right, swap the loops so the column loop is on the outside:

```
for (int c = 0; c < grid[0].length; c++) {  
    for (int r = 0; r < grid.length; r++) {  
        // visit grid[r][c]  
    }  
}
```

The loop on the outside moves slowest. Here the column index stays fixed while the row index runs down the whole column, so for `{{1, 2, 3}, {4, 5, 6}}` the visit order is 1, 4, 2, 5, 3, 6. Compare that with row-major order, which is 1, 2, 3, 4, 5, 6. When a tracing question gives you nested loops, the first thing to check is which index the outer loop controls.

Java also offers the **enhanced for loop**, nested the same way:

```
for (int[] row : grid) {  
    for (int val : row) {  
        System.out.print(val + " ");  
    }  
}
```

This visits every element in row-major order. Notice the outer variable is `int[]`, a whole row, because the elements of a 2D array are arrays. Two limits to remember. The loop gives you no index, so you cannot know positions or stop at a particular cell. And `val` is a copy of each element, so assigning to it changes nothing in the grid.

Trap. `for (int val : grid)` does not compile. The outer loop must declare an `int[]` because each element of `grid` is a row array. If an enhanced for loop over a 2D array looks odd, check the type of the outer variable first.

8.2 Common 2D Array Algorithms

Most 2D array algorithms are the 1D patterns you already know, with one more loop wrapped around them. Summing every element uses an **accumulator**:

```
int[][] vals = { {3, 1, 4}, {1, 5, 9} };
int total = 0;
for (int r = 0; r < vals.length; r++) {
    for (int c = 0; c < vals[r].length; c++) {
        total += vals[r][c];
    }
}
// total is 23
```

Finding the maximum follows the same shape, but initialize from the grid, not from 0:

```
int max = arr[0][0];
for (int r = 0; r < arr.length; r++) {
    for (int c = 0; c < arr[r].length; c++) {
        if (arr[r][c] > max) {
            max = arr[r][c];
        }
    }
}
```

Starting from `arr[0][0]` matters because the grid might hold all negative numbers, in which case a starting value of 0 would be reported as the maximum even though 0 is not in the grid. The same caution applies to minimums.

When each row needs its own result, compute one row at a time with a helper and reset the accumulator for every row:

```
public static int rowSum(int[][] arr, int row) {
    int sum = 0;
    for (int c = 0; c < arr[row].length; c++) {
        sum += arr[row][c];
    }
    return sum;
}

public static int rowWithMaxSum(int[][] arr) {
    int bestRow = 0;
    for (int r = 1; r < arr.length; r++) {
        if (rowSum(arr, r) > rowSum(arr, bestRow)) {
            bestRow = r;
        }
    }
    return bestRow;
}
```

For `{{2, 2}, {5, 1}, {3, 3}}` the row sums are 4, 6, and 6, and the method returns 1, the first row with the largest sum. This row-at-a-time structure is the shape of FRQ 4, which asks you to write methods that process a grid under some condition.

Trap. The per-row accumulator must be declared and reset inside the outer loop. If `sum` is declared before both loops, row 1's sum quietly includes row 0's values, and every row after the first is wrong. On FRQ 4 this single misplaced line costs the point for correct accumulation. The other classic is `r <= arr.length`, which reaches one index past the last row and throws an `ArrayIndexOutOfBoundsException`.

Confusions That Cost Points

PAIR	HOW TO KEEP THEM STRAIGHT
<code>arr[r][c]</code> vs <code>arr[c][r]</code>	The first index is the row and the second is the column, always. Swapping them reads the wrong cell on a square grid and throws <code>ArrayIndexOutOfBoundsException</code> on a non-square one.
<code>arr.length</code> vs total elements	<code>arr.length</code> is the number of rows. Total elements is rows times columns, never <code>arr.length</code> alone.
<code>arr[r].length</code> vs <code>arr.length</code> as the inner bound	The inner loop runs over columns, so its bound is the row's length, <code>arr[r].length</code> . Using <code>arr.length</code> for both loops only works on square grids.
Row-major vs column-major	The outer loop decides the order. Row loop outside visits left to right, top to bottom. Column loop outside visits top to bottom, left to right.
Enhanced for vs indexed loop	Enhanced for visits values in row-major order but gives no indices and cannot change elements. When you need positions or must modify the grid, use indexed loops.
Accumulator inside vs outside the outer loop	A per-row sum must be declared and reset inside the outer loop. Declared outside, it carries one row's total into the next and every later result is wrong.

Practice Questions

Original questions written for this guide in the style of the AP exam. Answers and explanations are on the next page, so complete the questions before checking them.

1. Consider the declaration `int[][] grid = new int[3][4];`. Which statement is true?

1. `grid` has 3 rows and 4 columns, and `grid.length` is 3
2. `grid` has 4 rows and 3 columns, and `grid.length` is 12
3. `grid` has 3 rows and 4 columns, and `grid.length` is 12
4. `grid` has 4 rows and 3 columns, and `grid.length` is 3

2. What is printed by the following code?

```
int[][] a = { {1, 2, 3}, {4, 5, 6} };
for (int r = 0; r < a.length; r++) {
    for (int c = 0; c < a[r].length; c++) {
        if (a[r][c] % 2 == 0)
            System.out.print(a[r][c]);
    }
}
```

1. 246
2. 24
3. 46
4. 123456

3. What is printed by the following code?

```
int[][] b = { {1, 2}, {3, 4}, {5, 6} };
for (int c = 0; c < b[0].length; c++) {
    for (int r = 0; r < b.length; r++) {
        System.out.print(b[r][c] + " ");
    }
}
```

1. 1 2 3 4 5 6
2. 1 3 5 2 4 6
3. 5 3 1 6 4 2
4. 2 4 6 1 3 5

4. What is printed by the following code?

```
int[][] d = { {2, 7}, {4, 9}, {6, 3} };
int count = 0;
for (int[] row : d) {
    for (int n : row) {
        if (n % 2 == 1)
            count++;
    }
}
System.out.println(count);
```

- 1.2
- 2.3
- 3.6
- 4.0

5. `vals` is a 3-by-5 rectangular int array. Which nested loop visits every element of `vals` exactly once?

1. `for (int r = 0; r < vals.length; r++)` with inner `for (int c = 0; c < vals.length; c++)`
2. `for (int r = 0; r < vals.length; r++)` with inner `for (int c = 0; c < vals[r].length; c++)`
3. `for (int r = 0; r < vals[0].length; r++)` with inner `for (int c = 0; c < vals.length; c++)`
4. `for (int r = 0; r <= vals.length; r++)` with inner `for (int c = 0; c <= vals[r].length; c++)`

6. What is printed by the following code?

```
int[][] e = { {10, 20, 30}, {40, 50, 60} };
int sum = 0;
for (int r = 0; r < e.length; r++)
    sum += e[r][1];
System.out.println(sum);
```

1. 70
2. 90
3. 60
4. 210

7. What is printed by the following code?

```
int[][] f = { {1, 2}, {3, 4, 5}, {6} };
int total = 0;
for (int r = 0; r < f.length; r++)
    total += f[r].length;
System.out.println(total);
```

1. 3
2. 6
3. 9
4. 5

8. `a` is a 3-by-5 rectangular int array. Which code segment correctly doubles every element of `a`?

1. `for (int r = 0; r < a.length; r++) { for (int c = 0; c < a[r].length; c++) { a[r][c] *= 2; } }`
2. `for (int[] row : a) { for (int val : row) { val *= 2; } }`
3. `for (int r = 0; r <= a.length; r++) { for (int c = 0; c <= a[r].length; c++) { a[r][c] *= 2; } }`
4. `for (int r = 0; r < a.length; r++) { for (int c = 0; c < a[r].length; c++) { a[c][r] *= 2; } }`

Answer Key

- 1. A.** In `new int[3][4]` the first dimension is rows, so the grid has 3 rows and 4 columns, and `grid.length` counts rows, which is 3. B swaps rows and columns and then treats `grid.length` as the element count. C gets the shape right but makes the same length mistake as B. D reads `grid.length` correctly as 3 but swaps the shape.
- 2. A.** Row 0 contributes 2 (1 and 3 are odd), and row 1 contributes 4 and 6 (5 is odd), so the output is 246. B traces only the first row. C traces only the second row. D ignores the even-only condition and prints every element.
- 3. B.** The column loop is outside, so column 0 is visited top to bottom (1, 3, 5) and then column 1 (2, 4, 6), giving 1 3 5 2 4 6. A is the row-major order, the result of reading the loops as if the row loop were outside. C reads each column bottom to top. D visits the second column before the first.
- 4. B.** The odd values are 7, 9, and 3, so count is 3. The enhanced for loops visit every element in row-major order. A misses one of the three odd values. C counts all six elements and ignores the condition. D assumes the enhanced for loop visits nothing, but it visits every element.
- 5. B.** The outer loop runs over the 3 rows and the inner loop runs over each row's 5 columns via `vals[r].length`. A uses `vals.length` (3) for the inner loop, so it visits only 3 of the 5 columns in each row and misses elements. C swaps the bounds, treating 5 as the row count and 3 as the column count. D uses `<=`, which reaches index `vals.length` and throws an `ArrayIndexOutOfBoundsException`.
- 6. A.** The loop fixes the column at 1 and walks the rows: `e[0][1]` is 20 and `e[1][1]` is 50, so the sum is 70. B sums column 2 (30 + 60 = 90), the result of misreading the fixed index. C sums row 0 (10 + 20 + 30 = 60), the result of fixing the row instead of the column. D sums the whole grid (210), the result of adding every element instead of one column.
- 7. B.** The rows have lengths 2, 3, and 1, so the total is 6. A answers `f.length`, the row count, confusing rows with total elements. C assumes a 3-by-3 square grid. D miscounts, for example by dropping the last row's single element.
- 8. A.** The indexed assignment `a[r][c] *= 2` writes through to the grid. B compiles but `val` is a copy of each element, so nothing in the grid changes. C reaches `a[3]` because of `<=` and throws an `ArrayIndexOutOfBoundsException`. D writes `a[c][r]` with `c` reaching 4 on a 3-row grid, which throws an `ArrayIndexOutOfBoundsException`.

When you check your answers, note which distinction each miss came from. Make a flashcard for that distinction and drill it spaced out over the next few days instead of rereading the whole section. If you missed one of these questions, the same distinction is worth practicing again in Rycal, where the 2D Array deck has flashcards for it and more practice questions use the same kinds of traps.

One-Page Recall Check

Say each answer out loud before you look back, and mark the ones you cannot finish. Anything you cannot say out loud yet belongs in your flashcard deck. In Rycal, add those items to the 2D Array deck and let spaced review bring them back over the next few days.

- Declare a 2D array with `new int[3][4]` and with an initializer list, and state the number of rows and columns each one creates.
- Explain what `arr.length` returns and what `arr[r].length` returns.
- Explain why the first index is the row and the second is the column.
- Write the standard row-major nested loop from memory.
- Write the column-major nested loop and explain how the visit order differs.
- Trace a nested loop over a 2-by-3 grid by hand and state the visit order.
- Explain what an enhanced for loop gives you on a 2D array, and name two things it cannot do.
- Explain why a per-row accumulator must be declared and reset inside the outer loop.
- State what goes wrong when rows and columns are swapped in the index, the loop bounds, and the declaration.
- Write code that finds the maximum value in a 2D array, and explain the starting value.

Where to go next. Turn every missed item above into flashcards and drill them spaced out over several days rather than in one sitting. In Rycal, open the 2D Array deck under AP Computer Science A. The deck covers the terms in this guide, and its practice questions target the same traps named here. If you have a test date, add it in the Test Planner. You can also start your next review with a Brain Dump, then check what you missed against this guide.

Key terms for this unit

2D array, Row, Column, `arr.length`, `arr[r].length`, Initializer list, Ragged array, Row-major order, Column-major order, Nested loop, Enhanced for loop, Traversal, Accumulator.

About this guide. Written for Rycal and aligned to the College Board AP Computer Science A course framework, Unit 8. All questions and explanations are original Rycal writing. Rycal is independent and is not affiliated with or endorsed by the College Board.