

4.11 2D Array Creation and Access

Self-Study Guide · About 45 minutes

By the end you will be able to

- Explain why Java stores a 2D array as an array of arrays, and picture one as a grid of rows and columns.
- Create a 2D array with an initializer list or with `new`, and say what every element holds right after `new` runs.
- Read and change one element with two indices, row first and column second.
- Get a whole row as a 1D array and explain why changing that row changes the 2D array.
- Find the number of rows and columns with `length`, name the valid indices, and predict when an index causes an `ArrayIndexOutOfBoundsException`.

What to remember from earlier

- An array holds a fixed number of values of one type. ([Topic 4.3: Array Creation and Access, 4.3.1 Arrays Store Many Values of One Type](#))
- `new int[5]` fills every element with a default value, such as 0 for `int` and `null` for references. ([Topic 4.3: Array Creation and Access, 4.3.3 New Arrays Start with Default Values](#))
- An initializer list such as `{90, 85, 77}` creates an array with those values. ([Topic 4.3: Array Creation and Access, 4.3.4 Creating an Array with an Initializer List](#))
- Valid indices are 0 through `length - 1`; any other index throws an `ArrayIndexOutOfBoundsException`. ([Topic 4.3: Array Creation and Access, 4.3.6 Valid Indexes and Out-of-Bounds Errors](#))
- Calling a method on `null` throws a `NullPointerException`. ([Topic 1.14: Calling Instance Methods, 1.14.5 Calling a Method on null](#))

4.11.1 A Grid Is an Array of Arrays

A teacher keeps quiz scores for three students, Ana, Ben, and Cy, on four quizzes each. On paper that is a small table: one line per student, one box per quiz. With the 1D arrays from Topic 4.3 you could make one array per student:

```
1 int[] anaScores = {90, 85, 77, 92};
2 int[] benScores = {68, 74, 81, 88};
3 int[] cyScores = {95, 91, 89, 100};
```

That works for three students, but a class of thirty would need thirty variables with thirty names, and there would be no single variable that holds "the whole class." What the teacher wants is one variable for the whole table.

Here is the key observation. A 1D array can hold any type of element: `int` values, `String` references, and so on. An array is itself an object, so a 1D array can also hold references to *other arrays*. Put the three student arrays into one outer array, and that outer array holds the whole table. Java calls this a **two-dimensional array**, or **2D array**: an array whose elements are

themselves 1D arrays of the same type. That is why people describe a 2D array as an **array of arrays**.

You can picture it in two ways. The first is a grid. Each inner array is a **row**, one line across the grid. The boxes at the same position in every row line up into a **column**, one line going down. Rows and columns are both numbered from 0, just like 1D array indices.

	Col 0	Col 1	Col 2	Col 3
Row 0	90	85	77	92
Row 1	68	74	81	88
Row 2	95	91	89	100

Row 0 is Ana's scores, row 1 is Ben's, and row 2 is Cy's. Column 2 holds everyone's score on the third quiz: 77, 81, and 89.

The second picture shows how Java actually stores it. The variable refers to one outer array with three elements, and each of those elements refers to a row array:

```
scores --> outer array
      index 0 --> [ 90 | 85 | 77 | 92 ]
      index 1 --> [ 68 | 74 | 81 | 88 ]
      index 2 --> [ 95 | 91 | 89 | 100 ]
```

The grid picture is the one you will use most when reading code. The array-of-arrays picture explains *why* the rules in the rest of this chapter work the way they do, so keep both in mind.

KEY IDEA

A 2D array is an array whose elements are 1D arrays. Each inner array is one row of a grid, and the elements at the same position in every row form a column.

Declaring and filling a 2D array

The type of a 2D array has two pairs of square brackets. `int[]` means "array of `int`," so `int[][]` means "array of `int[]`," an array of arrays of `int`. The fastest way to create a small 2D array is with an initializer list, just as in Topic 4.3. The difference is that each element of the outer list is itself an initializer list for one row.

WORKED EXAMPLE Building the quiz table

Read this declaration one brace at a time. It is spread over three lines only so the rows line up; Java would accept it on one line.

```
1 int[][] scores = { {90, 85, 77, 92},
2                   {68, 74, 81, 88},
3                   {95, 91, 89, 100} };
```

Piece	What it means
<code>int[][]</code>	The type: an array whose elements are <code>int[]</code> arrays
<code>scores</code>	The variable, which will hold a reference to the outer array
outer <code>{ ... }</code>	The outer array. It has three elements separated by commas, so 3 rows

{90, 85, 77, 92}	The first element of the outer array: row 0, a 1D array of four <code>int</code>
{68, 74, 81, 88}	Row 1
{95, 91, 89, 100}	Row 2

Java counts the inner lists to get the number of rows (3) and counts the values inside a row to get the number of columns (4). The result is the 3-by-4 grid pictured above. People always say the number of rows first: "a 3-by-4 array" has 3 rows and 4 columns.

A 2D array can hold object references as well as primitives. Change one thing, the element type, and the same pattern builds a seating chart of names:

```
1 String[][] seats = { {"Ana", "Ben", "Cy"},
2                     {"Dee", "Eli", "Fay"} };
```

The type is `String[][]`, an array of `String[]` rows. There are two inner lists, so 2 rows, and each holds three names, so 3 columns. Row 1, column 0 is "Dee": count rows from 0 (row 0 is the first line, row 1 the second), then count across from 0. Each box holds a reference to a `String` object, exactly as the boxes of a `String[]` did in Topic 4.3.

Every row in these examples has the same number of elements, so the grid is a rectangle. That is always the case in this course. (Java can build a grid whose rows have different lengths, but such grids are outside AP CSA, and you will not see them on the exam.)

WATCH OUT

Declaring a 1D type for 2D data. A student writes the quiz table with only one pair of brackets on the type:

WrongDimensions.java

```
1 public class WrongDimensions
2 {
3     public static void main(String[] args)
4     {
5         int[] scores = { {90, 85, 77, 92}, {68, 74, 81, 88} };
6         System.out.println(scores[0]);
7     }
8 }
```

```
WrongDimensions.java:5: error: illegal initializer for int
    int[] scores = { {90, 85, 77, 92}, {68, 74, 81, 88} };
                    ^
```

`int[]` promises that each element of the outer list is an `int`. The first element is `{90, 85, 77, 92}`, which is a whole row, not an `int`, so Java refuses to compile and reports the problem at each inner list. The number of bracket pairs in the type must match the nesting of the braces: two levels of braces need `int[][]`. The fix is to write `int[][] scores = ...`. (The wording of compiler messages can differ a little between Java versions.)

PAUSE AND TRY**Question 1** Multiple choice

Which declaration creates a 2D array with 2 rows and 3 columns?

- A. `int[][] g = { {1, 2, 3}, {4, 5, 6} };`
- B. `int[][] g = { {1, 2}, {3, 4}, {5, 6} };`
- C. `int[] g = { {1, 2, 3}, {4, 5, 6} };`
- D. `int[][] g = {1, 2, 3, 4, 5, 6};`

Question 2 Short answer

Using the picture of how Java stores `scores` (the outer array with arrows to three rows), explain in one or two sentences why a 2D array is called an "array of arrays." How many elements does the outer array of `scores` have?

4.11.2 Reading and Changing One Element

The teacher now wants Ben's score on the third quiz. In a 1D array, one index picks one box. In a grid, one number is not enough: you have to say which row and which column. Think of a seat in a theater, which needs both a row and a seat number. "Seat 5" alone could be any of a dozen seats.

Java uses two pairs of square brackets, one index in each. The first index is the **row index** and the second is the **column index**:

```
1 scores[row][col]
```

The order is always row first, column second. The array-of-arrays picture shows why this works with no new rules. `scores[1]` follows index 1 of the outer array to Ben's row, which is an ordinary 1D array. Then `[2]` picks index 2 of that row, just like any 1D array access from Topic 4.3. So `scores[1][2]` is really two 1D accesses in a row, read left to right.

KEY IDEA

The expression `grid[r][c]` names the element in row `r`, column `c`. The first index always picks the row and the second always picks the column, and both count from 0.

WORKED EXAMPLE Reading scores from the table

Each print statement uses two indices. Before reading the output, find each element in the grid in 4.11.1.

```

1 int[][] scores = { {90, 85, 77, 92},
2                   {68, 74, 81, 88},
3                   {95, 91, 89, 100} };
4 System.out.println(scores[1][2]);
5 System.out.println(scores[2][0]);
6 System.out.println(scores[0][3]);

```

Line	Expression	Row (first index)	Column (second index)	Value
4	scores[1][2]	1, Ben's row	2, the third quiz	81
5	scores[2][0]	2, Cy's row	0, the first quiz	95
6	scores[0][3]	0, Ana's row	3, the fourth quiz	92

```

81
95
92

```

Notice that the column index on line 4 is 2, but it names the *third* quiz. Counting from 0 works in both directions, exactly as with 1D arrays.

`scores[r][c]` is a variable, not just a value. Anywhere you could put a plain `int` variable, you can put `scores[r][c]`: on the left of `=` to store a new value, with compound assignment such as `+=` (Topic 1.6), or on the right side of an expression. This next example changes three elements.

```

1 int[][] scores = { {90, 85, 77, 92},
2                   {68, 74, 81, 88},
3                   {95, 91, 89, 100} };
4 scores[0][2] = 80;
5 scores[1][1] += 6;
6 scores[2][3] = scores[2][3] - scores[0][2];
7 System.out.println(scores[0][2]);
8 System.out.println(scores[1][1]);
9 System.out.println(scores[2][3]);

```

```

80
80
20

```

- Line 4 stores 80 in row 0, column 2. Ana's 77 is gone, replaced the same way any assignment replaces a value.
- Line 5 means `scores[1][1] = scores[1][1] + 6`. Ben's 74 becomes 80.
- Line 6 evaluates the right side first: `scores[2][3]` is 100 and `scores[0][2]` is now 80 (line 4 already changed it), so the result is 20, stored back in row 2, column 3.

Only the three named boxes change. Every other element keeps its value.

WATCH OUT

Swapping the row and column. A student wants Cy's score on quiz 1 (column 1) and writes:

```
1 int[][] scores = { {90, 85, 77, 92},  
2                   {68, 74, 81, 88},  
3                   {95, 91, 89, 100} };  
4 System.out.println("Cy, quiz 1: " + scores[1][2]);
```

Cy, quiz 1: 81

Cy's row is row 2, so the correct value is 91. The student wrote the indices in the order of the sentence, "quiz 1 for Cy," instead of the order Java uses. `scores[1][2]` is row 1 (Ben), column 2, which is 81. Java gave no error, because both indices happen to be valid. That is what makes this mistake dangerous: the program runs and prints a wrong number. The fix is to put the row first: `scores[2][1]`, which prints Cy, quiz 1: 91. When you write an access, say "row, then column" to yourself and check each index against the grid.

PAUSE AND TRY

Question 3 Tracing

Write the exact output. Draw the grid first, then update it line by line.

```
1 int[][] tiles = { {4, 7, 2},  
2                 {9, 5, 8} };  
3 tiles[1][0] = tiles[0][1] + tiles[1][2];  
4 tiles[0][2] *= 3;  
5 System.out.println(tiles[1][0]);  
6 System.out.println(tiles[0][2]);  
7 System.out.println(tiles[1][1] - tiles[0][0]);
```

Question 4 Multiple choice

Using the quiz table `scores` from this chapter, which expression gives Ben's score (row 1) on the last quiz (column 3), which is 88?

- A. `scores[1][3]`
- B. `scores[3][1]`
- C. `scores[2][3]`
- D. `scores[1, 3]`

4.11.3 Creating a 2D Array with new

An initializer list is handy when you already know every value. Often you do not. A theater that sells tickets needs a chart of 2 rows and 3 seats per row, with every seat starting out empty, and the program will fill seats in as people buy them. For that you need to create the grid at its full size first and put values in later.

In Topic 4.3 you created an empty 1D array with `new int[5]`. A 2D array uses the same keyword with two sizes:

```
1 boolean[][] taken = new boolean[2][3];
```

Piece	What it means
<code>boolean[][]</code>	The type of the variable: an array of <code>boolean[]</code> rows
<code>new</code>	Create a new array object
<code>boolean</code>	The element type, the type stored in each box
<code>[2]</code>	The number of rows (always first)
<code>[3]</code>	The number of columns, the length of every row

Java builds an outer array with 2 elements and gives it 2 row arrays of 3 elements each. Nobody has stored anything in those 6 boxes yet, but Java never leaves an element unset. When `new` creates a 2D array, every element starts with the **default value** for its type, the same defaults you met with 1D arrays:

Element type	Default value
<code>int</code>	0
<code>double</code>	0.0
<code>boolean</code>	false
any reference type, such as <code>String</code>	null

Java fills in defaults so that reading an element you have not yet assigned always gives a predictable answer instead of leftover garbage from memory.

KEY IDEA

`new type[rows][cols]` creates a grid with that many rows and columns, and every element starts at the default value for its type. The size is fixed when the array is created.

WORKED EXAMPLE Defaults in four kinds of grid

Each array below is created with `new` and never assigned, so every print shows a default.

```
1 int[][] counts = new int[2][3];
2 double[][] prices = new double[2][2];
3 boolean[][] taken = new boolean[3][4];
4 String[][] names = new String[2][5];
5 System.out.println(counts[1][2]);
6 System.out.println(prices[0][1]);
7 System.out.println(taken[2][3]);
8 System.out.println(names[0][4]);
```

```
0
0.0
false
null
```

- Line 1 makes a 2-by-3 grid of `int`. Line 5 reads row 1, column 2, the last box, and gets 0.
- Line 2 makes a 2-by-2 grid of `double`. The default prints as 0.0, because it is a `double`.
- Line 3 makes a 3-by-4 grid of `boolean`. Row 2, column 3 is `false`.
- Line 4 makes a 2-by-5 grid of `String`. Each box will someday hold a reference to a `String`, but right now it holds `null`, meaning "no object yet." Printing `null` shows the word `null`.

The sizes inside the brackets do not have to be literal numbers. Any `int` expression works, so a program can use variables. Here the theater stores its seat chart and sells two seats:

```
1 int rows = 2;
2 int seatsPerRow = 3;
3 boolean[][] taken = new boolean[rows][seatsPerRow];
4 taken[0][1] = true;
5 taken[1][2] = true;
6 System.out.println(taken[0][1]);
7 System.out.println(taken[0][2]);
8 System.out.println(taken[1][2]);
```

```
true
false
true
```

Line 3 creates a 2-by-3 grid because `rows` is 2 and `seatsPerRow` is 3. All six seats start `false`, meaning not taken. Lines 4 and 5 store `true` in two of them. Line 7 reads a seat that was never assigned, so it still holds its default, `false`.

Once the grid exists, its size cannot change. There is no statement that adds a row or makes the rows longer. This comes straight from 1D arrays: the outer array and every row are ordinary arrays, and an array's length is set when it is created. If the theater adds a row, the program must create a new, larger 2D array.

WATCH OUT

Calling a method on an empty `String` box. In a grid of a reference type, every box starts as `null`. A student stores one name and then asks for the length of a name in a box that was never filled:

NullSeat.java

```
1 public class NullSeat
2 {
3     public static void main(String[] args)
4     {
5         String[][] seats = new String[2][3];
6         seats[0][0] = "Ana";
7         System.out.println(seats[0][0].length());
8         System.out.println(seats[0][1].length());
9     }
10 }
```

```
3
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"String.length()" because ... is null
```

Line 7 works: `seats[0][0]` refers to "Ana", and its length is 3. Line 8 crashes. `seats[0][1]` was never assigned, so it holds `null`, and as you saw in Topic 1.14, calling a method on `null` throws a `NullPointerException`. Students expect an empty string, "", but `new` does not create any `String` objects at all; it only creates the boxes. (The exact message varies between Java versions.)

The fix is to check for `null` before calling a method, using an `if` statement from Topic 2.3:

NullSeatFixed.java

```
1 public class NullSeatFixed
2 {
3     public static void main(String[] args)
4     {
5         String[][] seats = new String[2][3];
6         seats[0][0] = "Ana";
7         System.out.println(seats[0][0].length());
8         if (seats[0][1] != null)
9         {
10             System.out.println(seats[0][1].length());
11         }
12         else
13         {
14             System.out.println("Seat is empty");
15         }
16     }
17 }
```

```
3
Seat is empty
```

PAUSE AND TRY

Question 5 Multiple choice

What does this code print?

```

1 boolean[][] lights = new boolean[2][2];
2 lights[0][1] = true;
3 System.out.println(lights[1][1]);

```

- A. false
- B. true
- C. null
- D. 0

Question 6 Tracing

Write the exact output.

```

1 double[][] fees = new double[3][2];
2 fees[2][1] = 12.5;
3 fees[0][0] = fees[2][1] * 2;
4 System.out.println(fees[0][0]);
5 System.out.println(fees[1][1]);
6 System.out.println(fees[2][1] + fees[2][0]);

```

4.11.4 One Row Is a 1D Array

Suppose a method already exists that works on one student's scores, an `int[]`, the kind of method you could have written in Topic 4.4. Can you hand it Ana's row from the 2D table, or do you have to copy her four scores into a new array first?

Look at the array-of-arrays picture again. Each element of the outer array is a 1D array. So using the 2D array's name with only one pair of brackets, `scores[1]`, gives you the element at index 1 of the outer array: Ben's whole row. Its type is `int[]`, and you can use it anywhere an `int[]` is allowed.

Expression	Brackets	What it names	Type
<code>scores</code>	none	the whole grid	<code>int[][]</code>
<code>scores[1]</code>	one	row 1, a 1D array	<code>int[]</code>
<code>scores[1][2]</code>	two	one element, row 1 column 2	<code>int</code>

Each pair of brackets peels off one level: from grid to row, then from row to a single number.

There is one more thing to remember from Topic 1.13 and Topic 4.3: arrays are objects, and a variable of an array type holds a reference. `scores[1]` does not make a copy of Ben's row. It gives

you a reference to the very row that lives inside the grid. If you store it in a variable, that variable and the grid share the row.

KEY IDEA

`grid[r]`, with one pair of brackets, is row `r` of the grid, a 1D array. It is the same row object that is inside the grid, not a copy, so changing an element through it changes the grid.

WORKED EXAMPLE Taking out Cy's row

Line 4 stores row 2 in a 1D array variable. Watch what line 7 does to the grid.

```
1 int[][] scores = { {90, 85, 77, 92},
2                   {68, 74, 81, 88},
3                   {95, 91, 89, 100} };
4 int[] cyRow = scores[2];
5 System.out.println(cyRow[1]);
6 System.out.println(cyRow.length);
7 cyRow[0] = 99;
8 System.out.println(scores[2][0]);
```

```
91
4
99
```

- Line 4: `scores[2]` is a reference to Cy's row. `cyRow` now refers to that same row array, `[95, 91, 89, 100]`.
- Line 5: `cyRow[1]` is an ordinary 1D access, index 1 of Cy's row: `91`. It is the same element as `scores[2][1]`.
- Line 6: `cyRow.length` is the length of the row, `4`, the number of columns.
- Line 7 stores `99` at index 0 of the row `cyRow` refers to.
- Line 8 reads `scores[2][0]`. `scores[2]` leads to the very row that line 7 changed, so it prints `99`, not `95`.

```
scores --> outer array
index 0 --> [ 90 | 85 | 77 | 92 ]
index 1 --> [ 68 | 74 | 81 | 88 ]
index 2 --> [ 99 | 91 | 89 | 100 ] <-- cyRow
```

There is only one row 2. `cyRow` and `scores[2]` are two ways to reach it.

The real payoff is passing a row to a method. The method `firstAndLast` below has an `int[]` parameter. It has no idea that its argument came from a 2D array.

RowToMethod.java

```
1 public class RowToMethod
2 {
3     public static String firstAndLast(int[] quizzes)
4     {
5         return quizzes[0] + " to " + quizzes[quizzes.length - 1];
6     }
7
8     public static void main(String[] args)
9     {
10         int[][] scores = { {90, 85, 77, 92},
11                             {68, 74, 81, 88},
12                             {95, 91, 89, 100} };
13         System.out.println("Ana: " + firstAndLast(scores[0]));
14         System.out.println("Ben: " + firstAndLast(scores[1]));
15     }
16 }
```

```
Ana: 90 to 92
Ben: 68 to 88
```

On line 13, the argument `scores[0]` is Ana's row, so the parameter `quizzes` gets a reference to that row (Topic 3.6). Inside the method, `quizzes.length` is 4, so `quizzes[quizzes.length - 1]` is `quizzes[3]`, the last score, 92. Line 14 passes Ben's row instead, and the same method reports 68 and 88. One method written for 1D arrays now works on any row.

WATCH OUT

Storing a row in an `int` variable. A student wants Ben's row and writes:

RowWrongType.java

```
1 public class RowWrongType
2 {
3     public static void main(String[] args)
4     {
5         int[][] scores = { {90, 85, 77, 92},
6                             {68, 74, 81, 88},
7                             {95, 91, 89, 100} };
8         int benRow = scores[1];
9         System.out.println(benRow);
10    }
11 }
```

```
RowWrongType.java:8: error: incompatible types: int[] cannot be converted to int
    int benRow = scores[1];
                   ^
```

With one pair of brackets, `scores[1]` is a whole row, an `int[]`, but `benRow` was declared to hold a single `int`. Students make this mistake because in a 1D array one pair of brackets always gives one element. In a 2D array, one pair gives a row and you need two pairs to reach

an element. Fix it by matching the type to what you want: `int[] benRow = scores[1];` for the row, or `int benScore = scores[1][2];` for one score.

PAUSE AND TRY

Question 7 Multiple choice

Given `double[][] temps = new double[3][5];`, what is the type of the expression `temps[2]`?

- A. `double[]`
- B. `double`
- C. `double[][]`
- D. `int`

Question 8 Tracing

Write the exact output.

```
1 int[][] stock = { {5, 0, 3},  
2                 {2, 8, 6} };  
3 int[] top = stock[0];  
4 top[1] = 4;  
5 stock[0][2] += top[0];  
6 System.out.println(stock[0][1]);  
7 System.out.println(top[2]);  
8 System.out.println(stock[1].length);
```

4.11.5 Counting Rows and Columns

A method that receives a 2D array often has no idea how big it is. The quiz table might have 3 students or 30. To reach the last student's last score, or just to avoid using an index that does not exist, code needs to ask the grid for its size.

A 1D array tells you its size through `length` (Topic 4.3). A 2D array has no separate "rows" or "columns" attribute; it does not need one, because the array-of-arrays picture gives both numbers:

- The outer array has one element per row, so `scores.length` is the **number of rows**.
- Each row is a 1D array whose length is the number of columns, so `scores[0].length` is the **number of columns**.

Row 0 is used because it is always there, and in this course every row has the same length, so any row would give the same answer.

The same counts give the valid indices. Like any array, indices run from 0 up to one less than the length:

Index	Valid values	For the quiz table
row index	0 to <code>scores.length - 1</code>	0, 1, 2
col index	0 to <code>scores[0].length - 1</code>	0, 1, 2, 3

An index outside these ranges throws an `ArrayIndexOutOfBoundsException`, the same exception as a bad index in a 1D array, and for the same reason: one of the two 1D accesses inside `grid[r][c]` asked for a box that does not exist.

KEY IDEA

For a 2D array `grid`, `grid.length` is the number of rows and `grid[0].length` is the number of columns. Valid row indices are 0 through `grid.length - 1`, valid column indices are 0 through `grid[0].length - 1`, and anything else throws an `ArrayIndexOutOfBoundsException`.

WORKED EXAMPLE Finding the last score

Lines 4 and 5 compute the last valid row and column from the lengths, instead of counting by hand.

```

1 int[][] scores = { {90, 85, 77, 92},
2                   {68, 74, 81, 88},
3                   {95, 91, 89, 100} };
4 int lastRow = scores.length - 1;
5 int lastCol = scores[0].length - 1;
6 System.out.println(scores.length);
7 System.out.println(scores[0].length);
8 System.out.println(scores[lastRow][lastCol]);

```

```

3
4
100

```

Line	Expression	Why	Value
4	<code>scores.length - 1</code>	3 rows, so the last row index is 3 - 1	2
5	<code>scores[0].length - 1</code>	row 0 has 4 elements, so last column is 3	3
6	<code>scores.length</code>	elements in the outer array, one per row	3
7	<code>scores[0].length</code>	elements in row 0	4
8	<code>scores[lastRow][lastCol]</code>	<code>scores[2][3]</code> , Cy's last quiz	100

If the teacher adds a fourth student row to the initializer list, lines 4 and 5 adjust by themselves, and line 8 still finds the bottom-right score.

Now change the shape. This grid, made with `new`, has more rows than columns.

```

1 int[][] board = new int[5][2];
2 board[board.length - 1][board[0].length - 1] = 7;
3 System.out.println(board.length);
4 System.out.println(board[0].length);
5 System.out.println(board[4][1]);

```

```

5
2
7

```

`new int[5][2]` makes 5 rows of 2 columns, so `board.length` is 5 and `board[0].length` is 2. Line 2 stores 7 at `board[4][1]`, the bottom-right box: row index 4 (5 - 1) and column index 1 (2 - 1). When a grid is not square, mixing up the two lengths gives a different answer, which the next Watch Out shows.

WATCH OUT

Using the wrong length. A student wants Ana's last quiz and uses `scores.length` for the column:

```

1 int[][] scores = { {90, 85, 77, 92},
2                   {68, 74, 81, 88},
3                   {95, 91, 89, 100} };
4 System.out.println("Ana's last quiz: " + scores[0][scores.length - 1]);

```

```
Ana's last quiz: 77
```

The answer should be 92. `scores.length` is the number of rows, 3, so `scores.length - 1` is 2, and the program read column 2. The quiz table has 3 rows but 4 columns, so the two lengths are not interchangeable. On a square grid the bug would hide, which is why it slips through. Use `scores[0].length - 1` for the last column.

The other common slip forgets the `- 1`:

PastLastColumn.java

```

1 public class PastLastColumn
2 {
3     public static void main(String[] args)
4     {
5         int[][] scores = { {90, 85, 77, 92},
6                           {68, 74, 81, 88},
7                           {95, 91, 89, 100} };
8         System.out.println("Ben's last quiz:");
9         System.out.println(scores[1][scores[0].length]);
10    }
11 }

```

```

Ben's last quiz:
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of
bounds for length 4

```

Line 9 asks for column `scores[0].length`, which is 4, but the columns are 0 through 3. Java throws an `ArrayIndexOutOfBoundsException` when the program runs. The message names the bad index (4) and the length of the array it was used on (Ben's row, length 4); the wording can vary between Java versions. Writing `scores[1][scores[0].length - 1]` prints 88.

PAUSE AND TRY

Question 9 Multiple choice

Given `String[][] menu = new String[4][6];`, which describes all the valid indices for `menu[r][c]`?

- A. `r` from 0 to 3 and `c` from 0 to 5
- B. `r` from 1 to 4 and `c` from 1 to 6
- C. `r` from 0 to 4 and `c` from 0 to 6
- D. `r` from 0 to 5 and `c` from 0 to 3

Question 10 Tracing

Write the exact output.

```
1 int[][] marks = new int[2][5];
2 marks[marks.length - 1][0] = marks[0].length;
3 marks[0][marks[0].length - 1] = marks.length * 10;
4 System.out.println(marks[1][0]);
5 System.out.println(marks[0][4]);
6 System.out.println(marks[1][4]);
```

Putting it together

This program runs a small ticket booth. It creates a seat chart with `new`, fills seats using indices, uses `length` to find the last row and column, changes a seat through a row variable, and checks for an empty seat before treating it as a name.

TicketBooth.java

```

1 public class TicketBooth
2 {
3     public static void main(String[] args)
4     {
5         String[][] seats = new String[3][4];
6         seats[0][0] = "Ana";
7         seats[1][3] = "Ben";
8         seats[seats.length - 1][0] = "Cy";
9         String[] middle = seats[1];
10        middle[1] = "Dee";
11        System.out.println("Rows: " + seats.length);
12        System.out.println("Seats per row: " + seats[0].length);
13        System.out.println("Row 1, seat 1: " + seats[1][1]);
14        System.out.println("Row 2, seat 0: " + seats[2][0]);
15        System.out.println("Last seat: " + seats[seats.length - 1][seats[0].length -
16        1]);
17        if (seats[0][2] == null)
18        {
19            System.out.println("Row 0, seat 2 is open");
20        }
21    }
22 }

```

The chart has 3 rows and 4 seats per row, and every box starts as `null`.

Line(s)	What happens	Seats filled so far
5	<code>new String[3][4]</code> makes 12 boxes, all <code>null</code>	none
6	Row 0, seat 0 gets "Ana"	[0][0] Ana
7	Row 1, seat 3 gets "Ben"	+ [1][3] Ben
8	<code>seats.length - 1</code> is 2, the last row; row 2, seat 0 gets "Cy"	+ [2][0] Cy
9	<code>middle</code> refers to row 1 itself, not a copy	(no change)
10	<code>middle[1]</code> is the same box as <code>seats[1][1]</code> ; it gets "Dee"	+ [1][1] Dee
11–12	<code>seats.length</code> is 3 rows; <code>seats[0].length</code> is 4 seats per row	
13	<code>seats[1][1]</code> was set through <code>middle</code>	
14	<code>seats[2][0]</code> was set on line 8	
15	<code>seats[2][3]</code> , the bottom-right seat, was never assigned, so it is <code>null</code>	
16–19	<code>seats[0][2]</code> is still <code>null</code> , so the condition is true	

```

Rows: 3
Seats per row: 4
Row 1, seat 1: Dee
Row 2, seat 0: Cy
Last seat: null
Row 0, seat 2 is open

```

Line 15 prints the word `null` by joining it to a string, which is safe; only calling a method on `null` would crash. Line 16 avoids that kind of crash by comparing with `null` before using the seat.

Common mistakes

Each of these mistakes was taught in its own subsection. This table lets you look up a message you see when your own code goes wrong.

What you see	Likely cause	Fix
illegal initializer for int (compile error)	A 2D initializer list stored in a 1D type such as <code>int[]</code>	Declare the variable as <code>int[][]</code>
incompatible types: <code>int[]</code> cannot be converted to <code>int</code>	One pair of brackets (a whole row) stored in an <code>int</code> variable	Use <code>int[]</code> for a row, or add the column index for one element
<code>ArrayIndexOutOfBoundsException</code> (runtime)	An index equal to a length, a missing <code>- 1</code> , or row and column swapped	Keep rows in 0 to <code>grid.length - 1</code> , columns in 0 to <code>grid[0].length - 1</code>
<code>NullPointerException</code> (runtime)	A method called on an unfilled box of a reference-type grid	Assign the box first, or check <code>!= null</code>
Wrong value, no error	Row and column swapped, or <code>grid.length</code> used for columns	Row first, then column; columns come from <code>grid[0].length</code>

Vocabulary

Term	Definition	Example
two-dimensional (2D) array	An array whose elements are 1D arrays of the same type, pictured as a grid	<code>int[][] scores</code>
array of arrays	How Java stores a 2D array: an outer array whose elements refer to row arrays	<code>scores</code> refers to 3 row arrays
row	One inner array of a 2D array, a line across the grid	<code>scores[1]</code> is <code>{68, 74, 81, 88}</code>
column	The elements at the same index in every row, a line down the grid	column 2 of <code>scores</code> is <code>77, 81, 89</code>
row index	The first index in <code>grid[r][c]</code> , which picks the row	the 1 in <code>scores[1][2]</code>
column index	The second index in <code>grid[r][c]</code> , which picks the column	the 2 in <code>scores[1][2]</code>
default value (review)	The value <code>new</code> puts in every element: <code>0</code> , <code>0.0</code> , <code>false</code> , or <code>null</code> depending on the type	<code>new int[2][3]</code> holds six 0s

Chapter summary

- A 2D array is an array of arrays: an outer array whose elements are 1D row arrays of the same type. Picture it as a grid of rows and columns, both numbered from 0. It can hold primitives or object references, and in this course every row has the same length.

- Create a 2D array with an initializer list of row lists, such as `{ {1, 2, 3}, {4, 5, 6} }`, or with `new type[rows][cols]`, which fills every element with its default value: `0`, `0.0`, `false`, or `null`. The size cannot change after creation.
- `grid[r][c]` reads or changes one element. The first index is always the row and the second is always the column.
- `grid[r]`, with one pair of brackets, is row `r` as a 1D array. It is the same row object as the one inside the grid, so changes made through it show up in the grid.
- `grid.length` is the number of rows and `grid[0].length` is the number of columns. Indices outside `0` to `length - 1` throw an `ArrayIndexOutOfBoundsException`.

Review questions

Question 11 Multiple choice

A weather program creates `double[][] rainfall = new double[4][7]`; to store 4 weeks (rows) of 7 days (columns). Which statement is true right after this line runs?

- A. `rainfall` has 7 rows and 4 columns.
- B. `rainfall[3][6]` holds `0.0`.
- C. `rainfall[4][7]` is the last element.
- D. `rainfall.length` is 28.

Question 12 Tracing

Write the exact output.

```
1 int[][] grid = { {3, 1, 4, 1},
2                 {5, 9, 2, 7} };
3 int[] second = grid[1];
4 grid[0][grid.length] = second[3];
5 second[0] -= grid[0][0];
6 System.out.println(grid[0][2]);
7 System.out.println(grid[1][0]);
8 System.out.println(grid.length + grid[1].length);
```

Question 13 Debugging

This program should print the bottom-right temperature, `69.0`. Instead it prints `Last reading:` and then crashes with an `ArrayIndexOutOfBoundsException`. Explain the error and correct line 8.

BottomRight.java

```
1 public class BottomRight
2 {
3     public static void main(String[] args)
4     {
5         double[][] temps = { {61.5, 70.0, 66.5},
6                               {58.0, 73.5, 69.0} };
7         System.out.println("Last reading:");
8         System.out.println(temps[temps.length][temps[0].length]);
9     }
10 }
```

Question 14 Short answer

A student asks: "Why is the number of columns `grid[0].length` and not something like `grid.columns`? And why row 0? Would `grid[1].length` work too?" Answer the student in two or three sentences, using the idea of an array of arrays.

Question 15 Multiple choice

What does this code print?

```
1 int[][] nums = { {2, 4},
2                  {6, 8},
3                  {10, 12} };
4 System.out.println(nums[2][1] + nums.length * nums[0].length);
```

- A. 18
- B. 12
- C. 24
- D. Nothing; it throws an `ArrayIndexOutOfBoundsException` because `nums` has only 2 columns

Question 16 Debugging

A school tracks absences for 3 school days (rows) and 4 class periods (columns), as the comment says. The program crashes with an `ArrayIndexOutOfBoundsException` before printing anything. Find the error and correct the line that causes it.

AttendanceChart.java

```
1 public class AttendanceChart
2 {
3     public static void main(String[] args)
4     {
5         // 3 school days (rows) by 4 class periods (columns)
6         int[][] absences = new int[4][3];
7         absences[0][1] = 2;
8         absences[2][3] = 1;
9         System.out.println(absences[0][1] + absences[2][3]);
10    }
11 }
```

Reflection

- ☐ I can explain why a 2D array is called an array of arrays and draw one as a grid of rows and columns. Reread: 4.11.1 A Grid Is an Array of Arrays
- ☐ I can read a 2D initializer list and say how many rows and columns it creates. Reread: 4.11.1 A Grid Is an Array of Arrays
- ☐ I can read and change one element with two indices, row first and column second. Reread: 4.11.2 Reading and Changing One Element
- ☐ I can create a 2D array with new and say what default value every element starts with. Reread: 4.11.3 Creating a 2D Array with new
- ☐ I can use one pair of brackets to get a row and explain why changing that row changes the grid. Reread: 4.11.4 One Row Is a 1D Array
- ☐ I can find the number of rows and columns with length and name every valid index. Reread: 4.11.5 Counting Rows and Columns