

4.12 2D Array Traversals

Self-Study Guide · About 50 minutes · With answer key

By the end you will be able to

- Use nested for loops to visit every element of a 2D array in row-major order, row by row.
- Use nested for loops to visit every element in column-major order, column by column.
- Write loop headers that visit a 2D array in other orders, such as backward or one row or column only.
- Traverse a 2D array with nested enhanced for loops and choose the correct type for each loop variable.
- Explain why assigning to an enhanced for loop variable does not change the array, and when an indexed loop is needed.
- Find and fix traversal bugs such as swapped bounds, swapped indexes, and an off-by-one starting index.

What to remember from earlier

- A 2D array is an array of arrays, and `grid[row][col]` reaches one element. ([Topic 4.11: 2D Array Creation and Access](#))
- `grid.length` is the number of rows and `grid[0].length` is the number of columns. ([Topic 4.11: 2D Array Creation and Access](#), [4.11.5 Counting Rows and Columns](#))
- A 1D array can be traversed with an indexed for loop or an enhanced for loop. ([Topic 4.4: Array Traversals](#))
- Assigning to an enhanced for loop variable changes only that variable. ([Topic 4.4: Array Traversals](#), [4.4.5 Assigning to the Loop Variable](#))
- The inner loop finishes all its iterations before the outer loop moves on. ([Topic 2.11: Nested Iteration](#), [2.11.2 The Inner Loop Finishes First](#))

4.12.1 Row-Major Order

A small movie theater keeps track of tickets sold in a 2D array: one row per day, one column per showtime. With what you know from Topic 4.11, you can read any single value, such as `tickets[1][2]`. But the manager wants to see the whole table printed, or to add 2 tickets to every entry after finding a stack of paper sales. Writing one statement per element would work for 6 elements and fail completely for 600. You need a way to **traverse** the 2D array: to visit every element, one at a time, in a planned order.

For a 1D array, one loop was enough (Topic 4.4): an index runs from `0` up to `length - 1`. A 2D array needs *two* indexes, a row and a column, so it needs two loops, one inside the other. This is exactly the nested iteration from Topic 2.11, and the rule you learned there decides the order: the inner loop finishes all of its iterations before the outer loop moves on.

Here is the most common plan. The outer loop picks a row. While that row is held steady, the inner loop walks across it, column by column. When the inner loop reaches the end of the row, the outer loop moves down to the next row and the inner loop starts over at column `0`. You read a page of a book the same way: across a line from left to right, then down to the next line.

Visiting a 2D array this way, across each row before moving down to the next row, is called **row-major order**.

KEY IDEA

Row-major order: the outer loop goes through the rows, and for each row the inner loop goes through every column. The whole first row is visited before any of the second row.

The pattern has a fixed shape. Each part has a job:

```
1 for (int row = 0; row < grid.length; row++)
2 {
3     for (int col = 0; col < grid[0].length; col++)
4     {
5         // use grid[row][col]
6     }
7 }
```

- `row < grid.length`: `grid.length` is the number of rows (the length of the outer array), so `row` takes every valid row index, 0 up to one less than the number of rows.
- `col < grid[0].length`: `grid[0]` is the first row, a 1D array, and its length is the number of columns. Every row of an AP-style 2D array has the same length, so `grid[row].length` gives the same number and also works.
- `grid[row][col]`: the row index always comes first, matching the order of the loops.
- `<`, not `<=`: the last valid index is one less than the length, exactly as in a 1D array.

WORKED EXAMPLE Printing the ticket table row by row

The array has 2 rows (days) and 3 columns (showtimes). Watch which loop uses which length.

```
1 int[][] tickets = { {12, 5, 9}, {7, 3, 10} };
2 for (int row = 0; row < tickets.length; row++)
3 {
4     for (int col = 0; col < tickets[0].length; col++)
5     {
6         System.out.print(tickets[row][col] + " ");
7     }
8     System.out.println();
9 }
```

Line 2 runs `row` over 0 and 1, because `tickets.length` is 2. Line 4 runs `col` over 0, 1, 2, because `tickets[0].length` is 3. Line 6 prints one element followed by a space, using `print` so the row stays on one line. Line 8 sits after the inner loop's closing brace, so it ends the line once per row.

row	col	tickets[row][col]	Printed so far
0	0	12	12
0	1	5	12 5
0	2	9	12 5 9
0	(3)	inner loop ends	line 8 ends line

1	0	7	7
1	1	3	7 3
1	2	10	7 3 10
1	(3)	inner loop ends	line 8 ends line

When `col` reaches 3, the check `col < 3` fails, the inner loop ends, and the outer loop moves to the next row. When `row` reaches 2, the outer check fails and the traversal is done.

```
12 5 9
7 3 10
```

Each line of output has a space after its last number, which you cannot see. The visiting order was `[0][0]`, `[0][1]`, `[0][2]`, `[1][0]`, `[1][1]`, `[1][2]`: the column index changes fastest.

A traversal does not have to print. Any statement that uses `grid[row][col]` runs once for each element, so the same loops can change every element. Because the code names the actual array location, the change is stored in the array.

WORKED EXAMPLE Adding 2 to every element

This time the inner condition uses `tickets[row].length`, the length of the current row. For this rectangular array it is 3 on every row, so it gives the same visits as `tickets[0].length`.

```
1 int[][] tickets = { {12, 5, 9}, {7, 3, 10} };
2 for (int row = 0; row < tickets.length; row++)
3 {
4     for (int col = 0; col < tickets[row].length; col++)
5     {
6         tickets[row][col] += 2;
7     }
8 }
9 System.out.println(tickets[0][0] + " " + tickets[0][2]);
10 System.out.println(tickets[1][0] + " " + tickets[1][2]);
```

The loops visit the six elements in the same row-major order as before, and line 6 adds 2 to each one (compound assignment, Topic 1.6). Lines 9 and 10 then print the first and last element of each row:

```
14 11
9 12
```

12 became 14, 9 became 11, 7 became 9, and 10 became 12. The middle elements changed too; they just were not printed.

WATCH OUT

Using the wrong length in a loop bound. It is easy to mix up which length belongs to which loop. Here the outer loop (for rows) uses the column count, and the inner loop (for columns) uses the row count:

BoundsSwapped.java

```
1 public class BoundsSwapped
2 {
3     public static void main(String[] args)
4     {
5         int[][] tickets = { {12, 5, 9}, {7, 3, 10} };
6         for (int row = 0; row < tickets[0].length; row++)
7         {
8             for (int col = 0; col < tickets.length; col++)
9             {
10                System.out.print(tickets[row][col] + " ");
11            }
12            System.out.println();
13        }
14    }
15 }
```

The program prints two lines and then crashes:

```
12 5
7 3
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 2 out of
bounds for length 2
```

(The exact wording of the message can vary between Java versions.) Line 6 lets `row` go up to 2, because `tickets[0].length` is 3. But there are only 2 rows, so when line 10 evaluates `tickets[2]`, there is no row with index 2. Notice also that before the crash, each line was missing its last column: the inner loop stopped at `col < 2`. On a square array (same number of rows and columns) this mistake would not show up at all, which is why it can hide in code until it meets a non-square array.

The fix is to match each loop to its own index: the loop whose variable goes in the first bracket uses `tickets.length`, and the loop whose variable goes in the second bracket uses `tickets[0].length`.

PAUSE AND TRY

Question 1 Tracing

Write the exact output of this code.

```
1 int[][] temps = { {61, 64}, {58, 70}, {66, 59} };
2 for (int day = 0; day < temps.length; day++)
3 {
4     System.out.print("Day " + day + ":");
5     for (int reading = 0; reading < temps[0].length; reading++)
6     {
7         System.out.print(" " + temps[day][reading]);
8     }
9     System.out.println();
10 }
```

Question 2 Multiple choice

What does this code print?

```

1 double[][] prices = new double[4][6];
2 int visits = 0;
3 for (int r = 0; r < prices.length; r++)
4 {
5     for (int c = 0; c < prices[0].length; c++)
6     {
7         visits++;
8     }
9 }
10 System.out.println(visits);

```

- A. 10
- B. 24
- C. 6
- D. 4

4.12.2 Column-Major Order

The theater manager now asks a different question: how did each showtime do across all the days? That means reading the table *down* each column: every value for showtime 0, then every value for showtime 1, and so on. Row-major order gives you the wrong grouping.

Switching to this order needs only one change: swap which loop is on the outside. If the outer loop picks a column and the inner loop walks down the rows, then the inner loop finishes a whole column before the outer loop moves right to the next one. Visiting a 2D array this way, down each column before moving to the next column, is called **column-major order**.

Two things do *not* change. The bounds still belong to their own index: the column loop still uses `grid[0].length`, and the row loop still uses `grid.length`. The element is still written `grid[row][col]`, row first, because the first bracket picks the row no matter which loop is on the outside.

KEY IDEA

Column-major order: the outer loop goes through the columns, and for each column the inner loop goes through every row. The element is still `grid[row][col]`; only the order of the loops changes.

Order	Outer loop	Inner loop	Element
Row-major	row from 0 to <code>grid.length</code>	col from 0 to <code>grid[0].length</code>	<code>grid[row][col]</code>
Column-major	col from 0 to <code>grid[0].length</code>	row from 0 to <code>grid.length</code>	<code>grid[row][col]</code>

(In both rows of the table, each loop stops before reaching the length.)

WORKED EXAMPLE Printing the ticket table column by column

Same array as in 4.12.1. Compare lines 2 and 4 with the row-major version: the headers traded places, but line 6 is identical.

```

1  int[][] tickets = { {12, 5, 9}, {7, 3, 10} };
2  for (int col = 0; col < tickets[0].length; col++)
3  {
4      for (int row = 0; row < tickets.length; row++)
5      {
6          System.out.print(tickets[row][col] + " ");
7      }
8      System.out.println();
9  }

```

col	row	tickets[row][col]	Printed so far
0	0	12	12
0	1	7	12 7
0	(2)	inner loop ends	line 8 ends line
1	0	5	5
1	1	3	5 3
1	(2)	inner loop ends	line 8 ends line
2	0	9	9
2	1	10	9 10
2	(2)	inner loop ends	line 8 ends line

```

12 7
5 3
9 10

```

The output has 3 lines of 2 numbers, the reverse of the row-major output's 2 lines of 3, because each printed line is now one column. The visiting order was `[0][0]`, `[1][0]`, `[0][1]`, `[1][1]`, `[0][2]`, `[1][2]`: this time the row index changes fastest.

The same six values come out in both orders; only the sequence differs. Here are both sequences for the ticket table side by side:

Order	Elements visited, in order
Row-major	12, 5, 9, 7, 3, 10
Column-major	12, 7, 5, 3, 9, 10

WORKED EXAMPLE Labeling each showtime

This version prints a label before each column, so line 3 is in the outer loop's body, before the inner loop.

```

1 int[][] tickets = { {12, 5, 9}, {7, 3, 10} };
2 for (int col = 0; col < tickets[0].length; col++)
3 {
4     System.out.print("Showtime " + col + ":");
5     for (int row = 0; row < tickets.length; row++)
6     {
7         System.out.print(" " + tickets[row][col]);
8     }
9     System.out.println();
10 }

```

```

Showtime 0: 12 7
Showtime 1: 5 3
Showtime 2: 9 10

```

Line 3 runs 3 times (once per column), and line 6 runs 6 times (once per element). Each line of output shows every day's sales for one showtime, which is the manager's question answered.

WATCH OUT

Swapping the indexes along with the loops. Because the column loop is now on the outside, students often write the column variable first inside the brackets too:

IndexesSwapped.java

```

1 public class IndexesSwapped
2 {
3     public static void main(String[] args)
4     {
5         int[][] tickets = { {12, 5, 9}, {7, 3, 10} };
6         for (int col = 0; col < tickets[0].length; col++)
7         {
8             for (int row = 0; row < tickets.length; row++)
9             {
10                System.out.print(tickets[col][row] + " ");
11            }
12            System.out.println();
13        }
14    }
15 }

```

```

12 5
7 3
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 2 out of
bounds for length 2

```

(Message wording can vary.) Line 10 uses `col` as the row index. When `col` is 0, it reads `tickets[0][0]` and `tickets[0][1]`, which is part of row 0, not column 0. When `col` reaches 2, it asks for `tickets[2]`, and there is no row 2. The first index in brackets means "row" no matter which loop is outside. Keep `tickets[row][col]`:

```
1 int[][] tickets = { {12, 5, 9}, {7, 3, 10} };
2 for (int col = 0; col < tickets[0].length; col++)
3 {
4     for (int row = 0; row < tickets.length; row++)
5     {
6         System.out.print(tickets[row][col] + " ");
7     }
8     System.out.println();
9 }
```

This is the correct column-major traversal and prints 12 7, 5 3, 9 10 on three lines.

PAUSE AND TRY

Question 3 Tracing

Write the exact output of this code. Notice that nothing ends a line until after both loops.

```
1 String[][] letters = { {"c", "a", "t"}, {"o", "w", "l"} };
2 for (int c = 0; c < letters[0].length; c++)
3 {
4     for (int r = 0; r < letters.length; r++)
5     {
6         System.out.print(letters[r][c]);
7     }
8 }
9 System.out.println();
```

Question 4 Multiple choice

What does this code print?

```
1 int[][] grid = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };
2 for (int c = 0; c < grid[0].length; c++)
3 {
4     for (int r = 0; r < grid.length; r++)
5     {
6         if (r == 1)
7         {
8             System.out.print(grid[r][c] + " ");
9         }
10    }
11 }
12 System.out.println();
```

- A. 2 6 10
- B. 5 6 7 8
- C. 1 5 9 2 6 10 3 7 11 4 8 12
- D. 5

4.12.3 Other Orders and Parts of the Grid

Row-major and column-major are the two standard orders, but they are not the only ones. A program might need the most recent day first, or just one day, or just one showtime. In every case, the order of the visits is decided by three things you control: which loop is outside, where each loop starts, and which way it counts. This is the same control you had over a single loop in Topic 2.8, now used twice.

To go backward through the rows, start at the last row index and count down to 0. The last valid index is one less than the length, so the last row is `grid.length - 1` and the last column is `grid[0].length - 1`. The condition becomes `>= 0` so that index 0 is still visited.

WORKED EXAMPLE Starting from the bottom-right corner

Both loops count down. Look at the starting value and condition in each header.

```
1 int[][] tickets = { {12, 5, 9}, {7, 3, 10} };
2 for (int row = tickets.length - 1; row >= 0; row--)
3 {
4     for (int col = tickets[0].length - 1; col >= 0; col--)
5     {
6         System.out.print(tickets[row][col] + " ");
7     }
8     System.out.println();
9 }
```

row	col values visited	Line printed
1	2, 1, 0	10 3 7
0	2, 1, 0	9 5 12

```
10 3 7
9 5 12
```

`row` starts at `tickets.length - 1`, which is 1, and stops after 0. For each row, `col` starts at 2 and counts down to 0. This is still a row-major traversal (a whole row is finished before the next one), just in reverse.

You can also visit only part of the grid. A single row is a 1D array, so one loop across the columns with the row index held fixed visits just that row. A single column needs one loop down the rows with the column index held fixed. No nested loop is needed, because only one index changes.

WORKED EXAMPLE One day and one showtime

Line 5 holds the row index at 1; line 11 holds the column index at 2.

```
1 int[][] tickets = { {12, 5, 9}, {7, 3, 10} };
2 System.out.print("Day 1:");
3 for (int col = 0; col < tickets[0].length; col++)
4 {
5     System.out.print(" " + tickets[1][col]);
6 }
7 System.out.println();
8 System.out.print("Showtime 2:");
9 for (int row = 0; row < tickets.length; row++)
10 {
11     System.out.print(" " + tickets[row][2]);
12 }
13 System.out.println();
```

```
Day 1: 7 3 10
Showtime 2: 9 10
```

The first loop visits `tickets[1][0]`, `tickets[1][1]`, `tickets[1][2]`: all of row 1, so it uses the column count as its bound. The second loop visits `tickets[0][2]` and `tickets[1][2]`: all of column 2, so it uses the row count. The labels use the indexes, so "Day 1" means row index 1, the second day.

KEY IDEA

You choose the order of a traversal by choosing which loop is outside, where each loop starts, and which direction it counts. To visit one row or one column, hold one index fixed and loop over the other.

WATCH OUT

Starting a backward loop at the length. When counting down, it is tempting to start at `tickets.length`, because that is the number you use as a bound when counting up:

BackwardOffByOne.java

```
1 public class BackwardOffByOne
2 {
3     public static void main(String[] args)
4     {
5         int[][] tickets = { {12, 5, 9}, {7, 3, 10} };
6         for (int row = tickets.length; row >= 0; row--)
7         {
8             for (int col = tickets[0].length - 1; col >= 0; col--)
9             {
10                 System.out.print(tickets[row][col] + " ");
11             }
12             System.out.println();
13         }
14     }
15 }
```

Nothing prints before the crash:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 2 out of bounds for length 2
```

(Wording can vary.) `tickets.length` is 2, so the very first visit on line 10 asks for row 2, one past the last row. Start at `tickets.length - 1`, the last valid index, as in the bottom-right example above.

PAUSE AND TRY

Question 5 Tracing

Write the exact output of this code.

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

Question 6 Multiple choice

This loop uses one index for both brackets. What does it print?

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

- A. 1 5 9
- B. 3 5 7
- C. 7 5 3
- D. 3 6 9

4.12.4 Nested Enhanced for Loops

Indexed loops give you full control, but when you just want every element in order and do not care about positions, all the index bookkeeping (`row < grid.length`, `grid[row][col]`) is extra work and extra places to make a mistake. For a 1D array, Topic 4.4 gave you a shortcut: the enhanced for loop, which hands you each element in turn with no index at all. A 2D array can use it too.

The key is the fact from Topic 4.11: a 2D array is an array of arrays. If an enhanced for loop goes through the outer array, each element it hands you is not a number; it is a whole row, which is a 1D

array. So the outer loop variable must have the type of a row. For an `int[][]`, that type is `int[]`. Then a second enhanced for loop can go through that row, and each element it hands you is a single `int`.

A **nested enhanced for loop** is an enhanced for loop inside the body of another enhanced for loop. For a 2D array, the outer loop goes through the rows and the inner loop goes through the elements of one row.

```
1 for (int[] row : grid)
2 {
3     for (int value : row)
4     {
5         // use value
6     }
7 }
```

- `int[] row`: the outer variable's type is one row of `grid`. It drops one pair of brackets from `int[][]`.
- `: grid`: the outer loop goes through the outer array, so it gets one row at a time.
- `int value`: the inner variable's type is the element type of the row, `int`. It drops the last pair of brackets.
- `: row`: the inner loop goes through the current row, not through `grid`.

Because the outer loop always goes through the rows from first to last, and the inner loop always goes across a row from first to last, a nested enhanced for loop always visits in **row-major order**. It cannot go column by column or backward; for those orders you need indexed loops.

KEY IDEA

In a nested enhanced for loop over a 2D array, the outer variable is a row, so its type is a 1D array such as `int[]`. The inner variable is one element of that row, so its type is the element type, such as `int`. The visits are always in row-major order.

WORKED EXAMPLE The ticket table with enhanced for loops

This produces the same output as the indexed row-major version in 4.12.1, with no indexes anywhere.

```
1 int[][] tickets = { {12, 5, 9}, {7, 3, 10} };
2 for (int[] day : tickets)
3 {
4     for (int count : day)
5     {
6         System.out.print(count + " ");
7     }
8     System.out.println();
9 }
```

Outer pass	day refers to	count values	Line printed
1	<code>tickets[0]</code>	12, 5, 9	12 5 9
2	<code>tickets[1]</code>	7, 3, 10	7 3 10

```
12 5 9
7 3 10
```

On the first outer pass, `day` refers to row 0, the array `{12, 5, 9}`. The inner loop hands `count` each value of that row. Line 8 ends the line, and then `day` moves to row 1. The variable names `day` and `count` describe what each one holds, which is easier to read than `row` and `value` when the data has a meaning.

The same shape works for any element type. Only the types in the headers change: for a `String[][]`, a row is a `String[]` and an element is a `String`.

WORKED EXAMPLE Initials from a seating chart

`seats` has 3 rows of 2 names. The inner loop prints the first letter of each name (`substring(0, 1)`, Topic 1.15), and line 8 marks the end of each row with a `/`.

```
1 String[][] seats = { {"Ana", "Ben"}, {"Cy", "Dee"}, {"Eli", "Fay"} };
2 for (String[] row : seats)
3 {
4     for (String name : row)
5     {
6         System.out.print(name.substring(0, 1));
7     }
8     System.out.print("/");
9 }
10 System.out.println();
```

```
AB/CD/EF/
```

Row 0 gives A and B, then `/`; row 1 gives C and D, then `/`; row 2 gives E and F, then `/`. Line 8 is in the outer loop's body, after the inner loop, so it runs once per row, exactly like the `println()` in the ticket example.

WATCH OUT

Giving the outer variable the element type. Because the array holds `int` values, students often write `int` in the outer header:

`OuterTypeWrong.java`

```
1 public class OuterTypeWrong
2 {
3     public static void main(String[] args)
4     {
5         int[][] tickets = { {12, 5, 9}, {7, 3, 10} };
6         for (int day : tickets)
7         {
8             System.out.println(day);
9         }
10    }
11 }
```

This does not compile:

```
OuterTypeWrong.java:6: error: incompatible types: int[] cannot be converted to int
```

(Wording can vary.) The elements of the *outer* array are rows, each an `int[]`, and an `int` variable cannot hold an array. Java's message says exactly that: it found an `int[]` where you promised an `int`. The fix is for `(int[] day : tickets)`, plus an inner loop to reach the individual numbers. The reverse mistake, writing `int[]` for the inner variable, fails the same way with the types flipped.

PAUSE AND TRY

Question 7 Tracing

Write the exact output of this code.

```
1 double[][] times = { {9.5, 8.75}, {10.0, 7.25} };
2 for (double[] lane : times)
3 {
4     for (double t : lane)
5     {
6         System.out.print(t + " ");
7     }
8     System.out.println();
9 }
```

Question 8 Multiple choice

`String[][] menu` holds the names of dishes, with one row per day. Which outer loop header correctly begins a nested enhanced for loop over `menu`?

- A. `for (String day : menu)`
- B. `for (String[] day : menu)`
- C. `for (String[][] day : menu)`
- D. `for (String[] day : menu[0])`

4.12.5 Changing Elements During a Traversal

Back to the theater: the manager wants to add 2 tickets to every entry. In 4.12.1 indexed loops did this. The enhanced for loop looks shorter, so it is tempting to use it here too. Will it work?

Recall the rule for 1D arrays from Topic 4.4: an enhanced for loop variable holds a *copy* of the current element. Assigning a new value to that variable changes the copy and leaves the array alone. The same rule holds for the inner variable of a nested enhanced for loop, because that variable holds a copy of one `int`.

WORKED EXAMPLE The bonus that disappears

Line 6 adds 2 to `count` six times. Then line 9 prints two elements of the array.

```

1 int[][] tickets = { {12, 5, 9}, {7, 3, 10} };
2 for (int[] day : tickets)
3 {
4     for (int count : day)
5     {
6         count += 2;
7     }
8 }
9 System.out.println(tickets[0][0] + " " + tickets[1][2]);

```

12 10

Nothing changed. On each inner pass, Java copies the current element into `count`; then `count += 2` changes `count` from 12 to 14, from 5 to 7, and so on. On the next pass, `count` gets a fresh copy of the next element, and the 14 is simply thrown away. The array itself was never assigned to.

Inner pass	Element in array	count after copy	count after line 6	Element afterward
1	12	12	14	12
2	5	5	7	5
...	...	...	...	...
6	10	10	12	10

KEY IDEA

Assigning a new value to an enhanced for loop variable changes only that variable, never the array. To change elements, assign to an array location such as `grid[row][col]`.

What about the outer variable? It works differently, and the difference explains a lot about arrays of arrays. A row is an array, which is an object, so the 2D array does not store the row itself; it stores a reference to it. The outer variable `day` gets a copy of that *reference*, which points to the very same row array that `tickets` points to. This is the reference sharing from Topic 3.6: two references to one object. So if you change an element *through* `day`, using `day[col]`, you change the real row.

WORKED EXAMPLE Changing elements through the row variable

The outer loop is still an enhanced for loop, but the inner loop now uses an index into `day`.

```

1 int[][] tickets = { {12, 5, 9}, {7, 3, 10} };
2 for (int[] day : tickets)
3 {
4     for (int col = 0; col < day.length; col++)
5     {
6         day[col] += 2;
7     }
8 }
9 System.out.println(tickets[0][0] + " " + tickets[1][2]);

```

14 12

This time the change sticks. On the first outer pass, `day` refers to the same array as `tickets[0]`, so `day[0] += 2` changes `tickets[0][0]` from 12 to 14. The same happens for every element of both rows, and `tickets[1][2]` becomes 12. The difference from the last example: `count += 2` assigned to a copied `int`, while `day[col] += 2` assigned to an element of the shared row.

Code inside the loops	Changes the array?	Why
<code>grid[row][col] += 2;</code>	Yes	assigns to an element of the array
<code>row[col] += 2; (outer enhanced)</code>	Yes	<code>row</code> refers to the same row array as <code>grid</code>
<code>value += 2; (inner enhanced)</code>	No	<code>value</code> is a copy of one element

WATCH OUT

Using an element's value as an index. An enhanced for loop hands you values, not positions. Students who want a position sometimes use the value in brackets:

ValueAsIndex.java

```

1 public class ValueAsIndex
2 {
3     public static void main(String[] args)
4     {
5         int[][] tickets = { {12, 5, 9}, {7, 3, 10} };
6         for (int[] day : tickets)
7         {
8             for (int count : day)
9             {
10                System.out.println(day[count]);
11            }
12        }
13    }
14 }
```

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 12 out of bounds for length 3

(Wording can vary.) On the first visit, `count` is 12, the first ticket count. Line 10 then asks for `day[12]`, but the row has only 3 elements. `count` is the element itself, so print `count`. If you need positions (to print them, to compare neighbors, or to visit in a different order), use indexed loops instead.

PAUSE AND TRY**Question 9** Short answer

A teacher stores quiz scores in `int[][] scores` and wants to reset every score to 0. She writes a nested enhanced for loop whose inner body is `score = 0;`, where `score` is the inner

loop variable. Afterward, the scores are unchanged. Explain why, and describe a loop that would work.

Question 10 Multiple choice

Each segment starts with `int[][] grid = { {1, 2}, {3, 4} }`. Which segments add 1 to every element of `grid`?

```
1 // I
2 for (int[] row : grid)
3 {
4     for (int value : row)
5     {
6         value += 1;
7     }
8 }
9
10 // II
11 for (int r = 0; r < grid.length; r++)
12 {
13     for (int c = 0; c < grid[0].length; c++)
14     {
15         grid[r][c] += 1;
16     }
17 }
18
19 // III
20 for (int[] row : grid)
21 {
22     for (int c = 0; c < row.length; c++)
23     {
24         row[c] += 1;
25     }
26 }
```

- A. I only
- B. II only
- C. II and III only
- D. I, II, and III

Putting it together

A school garden is planted in beds, recorded as a 2D array of plant names with "-" for an empty spot. The program below fills every empty spot with beans, then prints the garden two ways. It uses three traversals: an indexed row-major traversal to change elements, a nested enhanced for loop to print by row, and an indexed column-major traversal to print by column.

WORKED EXAMPLE Planting and printing a garden

The first traversal (lines 7–16) must change the array, so it uses indexes. The other two only read.

GardenGrid.java

```

1 public class GardenGrid
2 {
3     public static void main(String[] args)
4     {
5         String[][] garden = { {"kale", "-", "corn"}, {"-", "peas", "-"} };
6
7         for (int row = 0; row < garden.length; row++)
8         {
9             for (int col = 0; col < garden[0].length; col++)
10            {
11                if (garden[row][col].equals("-"))
12                {
13                    garden[row][col] = "beans";
14                }
15            }
16        }
17
18        System.out.println("By row:");
19        for (String[] bed : garden)
20        {
21            for (String plant : bed)
22            {
23                System.out.print(plant + " ");
24            }
25            System.out.println();
26        }
27
28        System.out.println("By column:");
29        for (int col = 0; col < garden[0].length; col++)
30        {
31            System.out.print("Column " + col + ":");
32            for (int row = 0; row < garden.length; row++)
33            {
34                System.out.print(" " + garden[row][col]);
35            }
36            System.out.println();
37        }
38    }
39 }

```

Lines 7–16, filling empty spots. The loops visit in row-major order. Line 11 compares each element with "-" using `equals` (Topic 1.15), and line 13 replaces the empty ones. This must be an indexed traversal: an enhanced loop variable is a copy, so assigning "beans" to it would change nothing.

Visit	[row][col]	Before	After
1	[0][0]	"kale"	"kale"
2	[0][1]	"-"	"beans"
3	[0][2]	"corn"	"corn"
4	[1][0]	"-"	"beans"

5	[1][1]	"peas"	"peas"
6	[1][2]	" - "	"beans"

Lines 19–26, printing by row. `bed` is a `String[]`, one row, and `plant` is a `String`. Each `bed` is printed on its own line.

Lines 29–37, printing by column. The outer loop goes through the 3 columns and the inner loop goes down the 2 rows, still reading `garden[row][col]`.

```
By row:
kale beans corn
beans peas beans
By column:
Column 0: kale beans
Column 1: beans peas
Column 2: corn beans
```

The printing traversals show the changed values, because the first traversal stored its changes in the array itself. Notice how each loop was chosen for its job: indexes to write, enhanced for a simple row-major read, and indexes again for an order the enhanced for loop cannot produce.

Vocabulary

Term	Definition	Example
traverse	To visit the elements of an array one at a time, in a planned order	printing every element of <code>tickets</code>
row-major order	Visiting a 2D array across each row, finishing one row before moving down to the next	<code>[0][0]</code> , <code>[0][1]</code> , <code>[0][2]</code> , <code>[1][0]</code> , ...
column-major order	Visiting a 2D array down each column, finishing one column before moving right to the next	<code>[0][0]</code> , <code>[1][0]</code> , <code>[0][1]</code> , <code>[1][1]</code> , ...
nested enhanced for loop	An enhanced for loop inside another; for a 2D array the outer variable is a row, the inner an element	<code>for (int[] row : grid) then</code> <code>for (int value : row)</code>

Chapter summary

- Traversing a 2D array takes nested loops. Each loop variable goes from 0 to one less than its length: `grid.length` for rows and `grid[0].length` for columns.
- In row-major order the outer loop goes through the rows and the inner loop across each row. In column-major order the outer loop goes through the columns and the inner loop down each column. Either way the element is `grid[row][col]`, row first.

- By choosing which loop is outside, where each loop starts, and which way it counts, you can visit in other orders, such as backward, or just one row or column with a single loop.
- In a nested enhanced for loop, the outer variable is a row, with a 1D array type such as `int[]`, and the inner variable is an element of that row. It always visits in row-major order.
- Assigning to an enhanced for loop variable does not change the array. To change elements, assign through an index, such as `grid[row][col]` or `row[col]`.
- Swapped bounds, swapped indexes, and starting a backward loop at the length all ask for an index that does not exist and throw an `ArrayIndexOutOfBoundsException`.

Review questions

Question 11 Tracing

This code chooses a direction for each row. Write the exact output.

```

1  int[][] grid = { {5, 1, 8}, {2, 7, 4}, {9, 3, 6} };
2  for (int r = 0; r < grid.length; r++)
3  {
4      if (r % 2 == 0)
5      {
6          for (int c = 0; c < grid[0].length; c++)
7          {
8              System.out.print(grid[r][c] + " ");
9          }
10     }
11     else
12     {
13         for (int c = grid[0].length - 1; c >= 0; c--)
14         {
15             System.out.print(grid[r][c] + " ");
16         }
17     }
18 }
19 System.out.println();

```

Question 12 Multiple choice

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

- `for (int r = 0; r < grid.length; r++)` with inner `for (int c = 0; c < grid[0].length; c++)`, printing `grid[r][c] + " "`
- `for (int c = 0; c < grid[0].length; c++)` with inner `for (int r = 0; r < grid.length; r++)`, printing `grid[r][c] + " "`
- `for (int c = 0; c < grid.length; c++)` with inner `for (int r = 0; r < grid[0].length; r++)`, printing `grid[r][c] + " "`
- `for (int[] row : grid)` with inner `for (int value : row)`, printing `value + " "`

Question 13 Tracing

The first traversal fills a new array; the second prints it. Write the exact output.

```

1 int[][] table = new int[2][3];
2 for (int r = 0; r < table.length; r++)
3 {
4     for (int c = 0; c < table[0].length; c++)
5     {
6         table[r][c] = r * 10 + c;
7     }
8 }
9 for (int c = 0; c < table[0].length; c++)
10 {
11     for (int r = 0; r < table.length; r++)
12     {
13         System.out.print(table[r][c] + " ");
14     }
15     System.out.println();
16 }

```

Question 14 Debugging

This code should print to be or not but does not compile. Find the error and correct it.

InnerTypeWrong.java

```

1 public class InnerTypeWrong
2 {
3     public static void main(String[] args)
4     {
5         String[][] words = { {"to", "be"}, {"or", "not"} };
6         for (String[] row : words)
7         {
8             for (String[] word : row)
9             {
10                 System.out.print(word + " ");
11             }
12         }
13         System.out.println();
14     }
15 }

```

Question 15 Multiple choice

What does this code print?

```

1 int[][] grid = { {4, 1}, {2, 8} };
2 for (int[] row : grid)
3 {
4     for (int value : row)
5     {
6         value = 0;
7     }
8     row[0] = 9;
9 }
10 System.out.println(grid[0][0] + " " + grid[0][1] + " " + grid[1][0] + " " + grid[1][1]);

```

A. 0 0 0 0

- B. 9 0 9 0
- C. 9 1 9 8
- D. 4 1 2 8

Question 16 Short answer

A classmate says, "Nested enhanced for loops are always better for 2D arrays because they are shorter." Give two situations from this chapter where you must use indexed loops instead, and explain why for each.

Reflection

- ☐ I can write the loop headers for a row-major traversal and say which length bounds each loop.
Reread: 4.12.1 Row-Major Order
- ☐ I can trace a row-major traversal and explain why swapped bounds throw an exception. Reread: 4.12.1 Row-Major Order
- ☐ I can change a row-major traversal into a column-major one and still write `grid[row][col]`.
Reread: 4.12.2 Column-Major Order
- ☐ I can trace a traversal that counts backward, or that visits only one row or one column. Reread: 4.12.3 Other Orders and Parts of the Grid
- ☐ I can write the types of both variables in a nested enhanced for loop and say what order it visits. Reread: 4.12.4 Nested Enhanced for Loops
- ☐ I can explain why assigning to an enhanced for loop variable does not change the array, and which loops do. Reread: 4.12.5 Changing Elements During a Traversal

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

```
Day 0: 61 64
Day 1: 58 70
Day 2: 66 59
```

`temps` has 3 rows and 2 columns. The outer loop runs `day` over 0, 1, 2. For each day, line 4 prints the label once, then the inner loop prints both readings of that row, each with a space in front, and line 9 ends the line. The label uses the row index, so the first line says `Day 0`.

Reread 4.12.1 *Row-Major Order* if your output differed.

Question 2

B. `prices` has 4 rows and 6 columns. The outer loop runs 4 times, and on each pass the inner loop runs 6 times, so `visits++` runs $4 \times 6 = 24$ times, once per element. A full traversal visits every element exactly once.

- **A** is wrong: it adds the loop counts ($4 + 6$), as if the inner loop ran only once in total. It starts over for every row.
- **C** is wrong: it counts only one row's worth of visits. The outer loop repeats the inner loop for all 4 rows.
- **D** is wrong: 4 counts only the passes of the outer loop. Every one of those passes runs the inner loop 6 times.

Reread 4.12.1 *Row-Major Order* if you chose differently.

Question 3

```
coawt1
```

`letters` has 2 rows and 3 columns, and the outer loop goes through the columns. Column 0 gives `c` (row 0) then `o` (row 1); column 1 gives `a` then `w`; column 2 gives `t` then `1`. Everything prints on one line with no spaces. Reading the same array in row-major order would give `catow1` instead.

Reread 4.12.2 *Column-Major Order* if your output differed.

Question 4

B. The loops traverse all 12 elements in column-major order, but line 8 runs only when `r` is 1, which is the second row. In each column, exactly one visit has `r` equal to 1: `grid[1][0]`, `grid[1][1]`, `grid[1][2]`, `grid[1][3]`, which are 5, 6, 7, 8.

- **A** is wrong: `2 6 10` is column 1, which is what you get if you treat the `r` in `r == 1` as a column index. `r` is the first index, so it picks a row.
- **C** is wrong: that is the full column-major sequence, which ignores the `if` statement.
- **D** is wrong: the `if` does not stop the loops. After printing 5, the traversal continues through the other columns, and each one has a row-1 element.

Reread 4.12.2 *Column-Major Order* if you chose differently.

Question 5

```
369
258
147
```

The outer loop goes through the columns, starting at the last one (`c` is 2) and counting down. For each column, the inner loop goes down the rows from top to bottom. Column 2 holds 3, 6, 9; column 1 holds 2, 5, 8; column 0 holds 1, 4, 7. Each column is printed on its own line.

Reread 4.12.3 *Other Orders and Parts of the Grid* if your output differed.

Question 6

B. `grid.length` is 3, so `k` takes 0, 1, 2. The column index is `3 - 1 - k`, which is 2, 1, 0. The visits are `grid[0][2]`, `grid[1][1]`, `grid[2][0]`: 3, 5, 7. Each visit moves down one row and left one column.

- **A** is wrong: 1 5 9 would come from `grid[k][k]`, where the column index counts up along with the row.
- **C** is wrong: it lists the right elements in reverse. `k` starts at 0, so the top row comes first.
- **D** is wrong: 3 6 9 is all of column 2, as if the column index stayed at 2. It changes with `k`.

Reread 4.12.3 *Other Orders and Parts of the Grid* if you chose differently.

Question 7

```
9.5 8.75
10.0 7.25
```

The outer variable `lane` is a `double[]`, one row. On the first pass it refers to `{9.5, 8.75}`, and the inner loop prints each value followed by a space; then line 8 ends the line. On the second pass `lane` refers to `{10.0, 7.25}`. A `double` always prints with a decimal point, so the third value prints as 10.0, not 10.

Reread 4.12.4 *Nested Enhanced for Loops* if your output differed.

Question 8

B. The outer loop goes through the outer array, whose elements are the rows. Each row of a `String[][]` is a `String[]`, so the variable's type must be `String[]`.

- **A** is wrong: `String` is the type of one dish, the *inner* variable's type. A row cannot be stored in a `String` variable, so this does not compile.
- **C** is wrong: `String[][]` is the type of the whole table, not of one row. This does not compile either.
- **D** is wrong: `menu[0]` is one row, whose elements are `Strings`, not `String[]`s. It would also go through only the first day.

Reread 4.12.4 *Nested Enhanced for Loops* if you chose differently.

Question 9

Model answer: The inner loop variable `score` holds a copy of each element. `score = 0;` changes the copy, and the next pass gets a new copy of the next element, so the array is never assigned to. To change the array, she must assign to an array location: use nested indexed for loops with

`scores[row][col] = 0;`, or keep the outer enhanced loop and use an inner indexed loop with `row[col] = 0;`, because the outer variable refers to the real row.

A complete answer says:

- the enhanced for loop variable is a copy of the element
- assigning to it does not change the array
- a working version assigns to an element through an index, such as `scores[row][col]`

Reread 4.12.5 *Changing Elements During a Traversal* if your explanation differed.

Question 10

C. Segment II assigns to `grid[r][c]`, an element of the array. Segment III assigns to `row[c]`, and `row` refers to the same row array that `grid` stores, so the elements change. Segment I assigns only to `value`, a copy, so `grid` stays `{ {1, 2}, {3, 4} }`.

- **A** is wrong: segment I is the one that does *not* work, because `value` is a copy of each element.
- **B** is wrong: it treats the outer enhanced variable as a copy too. It is a copy of a *reference*, so changing `row[c]` changes the real row.
- **D** is wrong: it includes segment I, which changes only its copies.

Reread 4.12.5 *Changing Elements During a Traversal* if you chose differently.

Question 11

5 1 8 4 7 2 9 3 6

r	r % 2 == 0?	Direction	Printed
0	true	left to right	5 1 8
1	false	right to left	4 7 2
2	true	left to right	9 3 6

The outer loop goes through the rows in order. Even-numbered rows (remainder 0 when divided by 2, Topic 1.3) use an inner loop that counts up from column 0; odd rows use one that starts at the last column, `grid[0].length - 1`, and counts down. Nothing ends a line until after the traversal, so all nine values print on one line. This is a custom order built from the same pieces as row-major order.

Reread 4.12.3 *Other Orders and Parts of the Grid* if your output differed.

Question 12

B. 1 4 2 5 3 6 goes down column 0 (1, 4), then column 1, then column 2: column-major order. That needs the column loop outside, bounded by the number of columns (`grid[0].length`, which is 3), and the row loop inside, bounded by the number of rows (`grid.length`, which is 2), still reading `grid[r][c]`.

- **A** is wrong: rows outside gives row-major order, 1 2 3 4 5 6.
- **C** is wrong: the loops are in column-major arrangement, but the bounds are swapped. The inner `r` goes up to 2, so after printing 1 and 4 it asks for `grid[2][0]` and throws an `ArrayIndexOutOfBoundsException`.
- **D** is wrong: a nested enhanced for loop always visits in row-major order, so it prints 1 2 3 4 5 6.

Reread 4.12.2 *Column-Major Order* and 4.12.4 *Nested Enhanced for Loops* if you chose differently.

Question 13

```
0 10
1 11
2 12
```

`new int[2][3]` makes 2 rows and 3 columns, all 0 at first (Topic 4.11). The first traversal stores `r * 10 + c` in every element, so the array becomes:

row	col 0	col 1	col 2
0	0	1	2
1	10	11	12

The second traversal is column-major: the outer loop takes each of the 3 columns, the inner loop goes down the 2 rows, and line 15 ends the line after each column. Column 0 is 0 10, column 1 is 1 11, column 2 is 2 12.

Reread 4.12.2 *Column-Major Order* if your output differed.

Question 14

Line 8 gives the inner variable the wrong type. The inner loop goes through `row`, which is a `String[]`, so each element it hands out is a single `String`. Declaring `word` as `String[]` asks Java to store a `String` in an array variable, and javac reports something like:

```
InnerTypeWrong.java:8: error: incompatible types: String cannot be converted to String[]
```

Corrected code: the inner variable's type is the element type, `String`.

```
1 String[][] words = { {"to", "be"}, {"or", "not"} };
2 for (String[] row : words)
3 {
4     for (String word : row)
5     {
6         System.out.print(word + " ");
7     }
8 }
9 System.out.println();
```

It prints `to be or not` (with a space after `not`). The outer header was already right: each element of `words` is a row, a `String[]`.

Reread 4.12.4 *Nested Enhanced for Loops* if you missed it.

Question 15

C. Inside the inner loop, `value = 0;` assigns to a copy of each element, so it changes nothing in `grid`. Line 8 is different: `row` refers to the real row array, so `row[0] = 9;` changes the first element of each row. `grid[0][0]` and `grid[1][0]` become 9, and the other two elements keep their values, 1 and 8.

- **A** is wrong: it believes `value = 0;` zeroes the array and ignores line 7.

- **B** is wrong: it believes `value = 0`; changes the array, and then `row[0] = 9` overwrites the first column. Only the `row[0]` assignment takes effect.
- **D** is wrong: it treats `row` as a copy too. The outer variable is a copy of a reference, so changes made through `row[0]` reach the real row.

Reread *4.12.5 Changing Elements During a Traversal* if you chose differently.

Question 16

Model answer: First, visiting in column-major order (or backward, or one column only) needs indexed loops, because a nested enhanced for loop's outer loop always hands out whole rows from first to last, so it can only produce row-major order. Second, changing every element, such as adding 2 to each ticket count, needs an assignment to an array location like `grid[row][col]`; assigning to the inner enhanced variable changes only a copy. (A third acceptable answer: whenever you need the positions themselves, such as printing a row or column number, because an enhanced loop gives values, not indexes.)

A complete answer gives two of these:

- a different order (column-major, backward, part of the grid) needs control of the indexes
- changing elements needs assignment through an index
- knowing an element's position needs an index

Reread *4.12.2 Column-Major Order*, *4.12.4 Nested Enhanced for Loops*, and *4.12.5 Changing Elements During a Traversal* if your answer differed.