2D Arrays

Overview

We know how to store data in an ordered sequence using arrays.

Our ordered collections have only been in one dimension so far, though.

Sometimes we want to represent data in multiple dimensions

- Images as grids of pixels, arrays at different points in time, matrices

Arrays of Arrays

We've seen arrays of...

- ints
- doubles
- Strings
- chars
- Songs

What about an array of *arrays*?

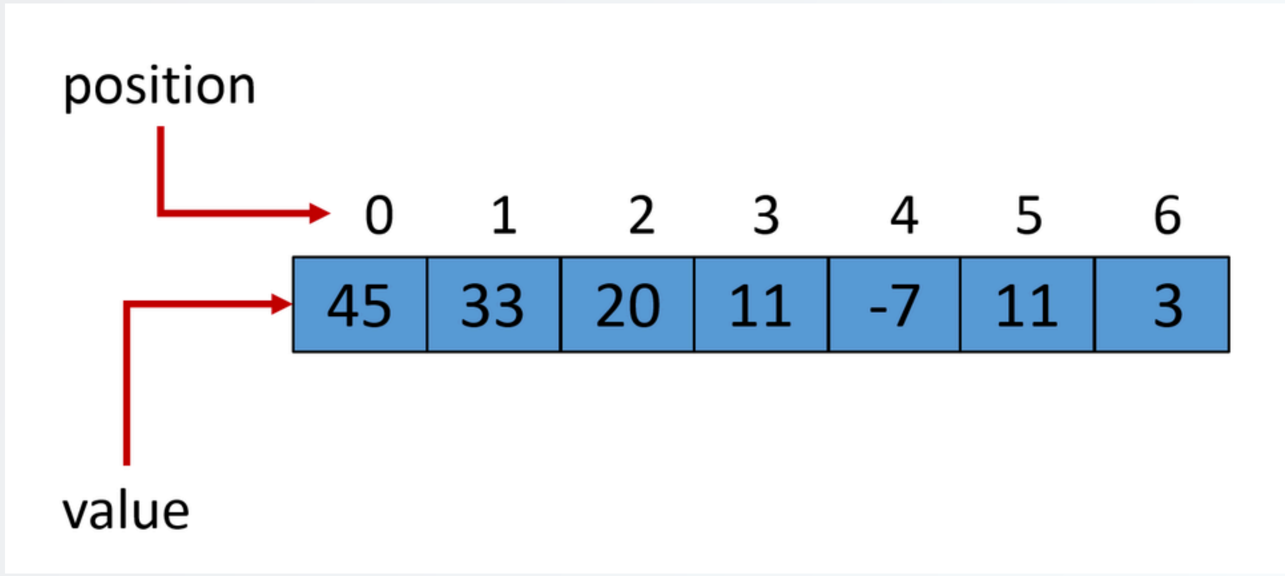
Learning Objectives

After this lecture, you should be able to...

- declare a 2D array
- initialize a 2D array with the new keyword or with an initializer list
- identify the dimensions of a 2D array
- access 2D array values
- modify 2D array values
- traverse a 2D array using the for-loop
- solve problems using 2D arrays

Two-Dimensional Arrays

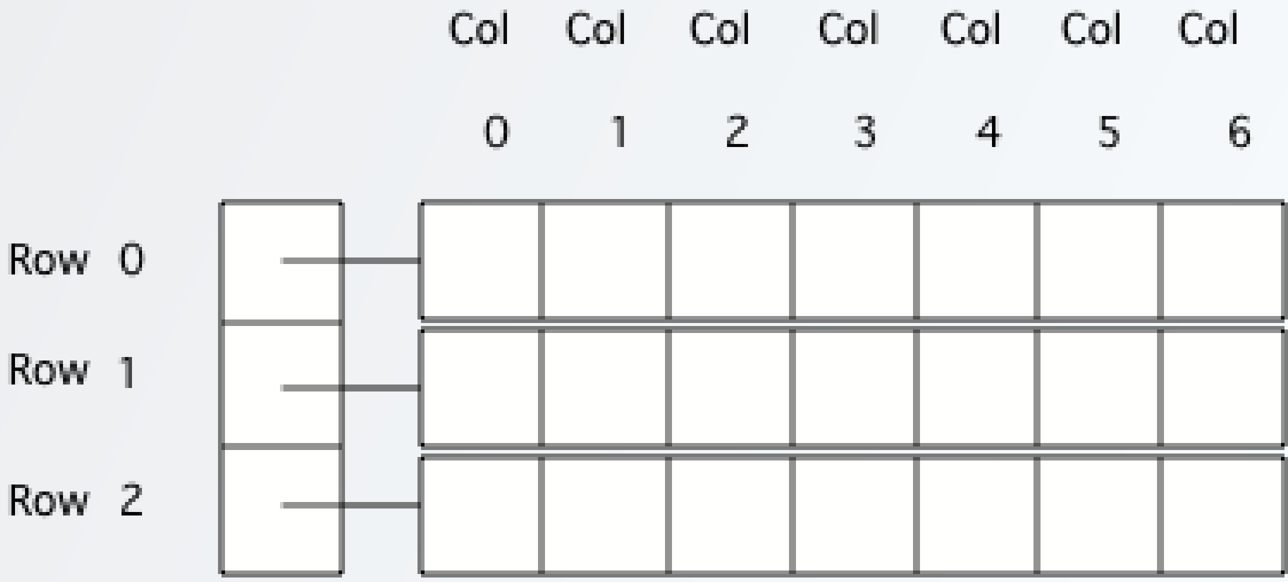
A **one-dimensional array** stores an ordered sequence of elements.



Two-Dimensional Arrays

A **two-dimensional array** can be thought of as a table of elements, with rows and columns.

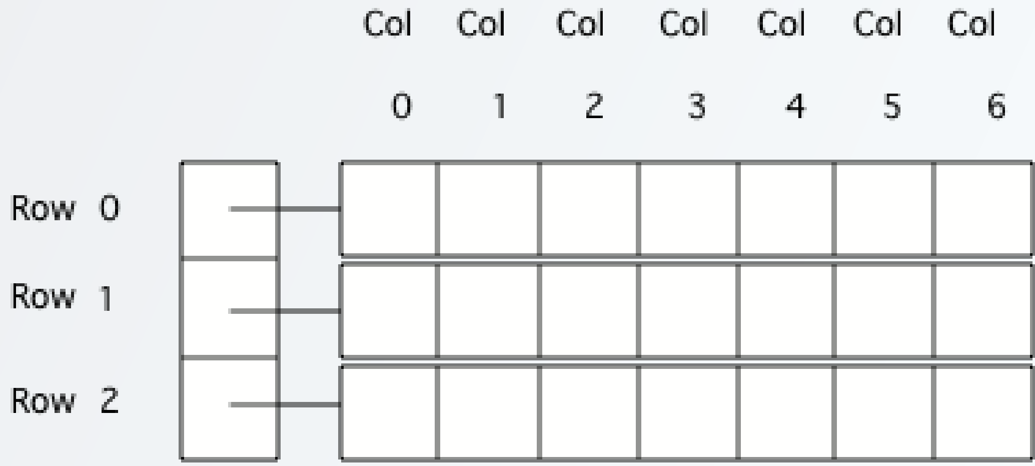
- The "outer" array represents the table, which consists of a bunch of rows.
- Each "inner" array is a row of the table, containing a bunch of cells (one per column.)



Declaration and Initialization

Here's one way to think about initializing a 2D array as an *array of int arrays*.

```
int[][] arrayOfArrays = new int[3][];
int[] firstArray = new int[7];
int[] secondArray = new int[7];
int[] thirdArray = new int[7];
arrayOfArrays[0] = firstArray;
arrayOfArrays[1] = secondArray;
arrayOfArrays[2] = thirdArray;
```



Declaration and Initialization

Here’s the general way to initialize a *rectangular* 2D array:

```
type[][] arrayName = new type[numRows][numCols];
```

- This creates a 2D array with numRows rows and numCols columns.

So, we could create the previous 2D array of ints with **three** rows and **seven** columns like so:

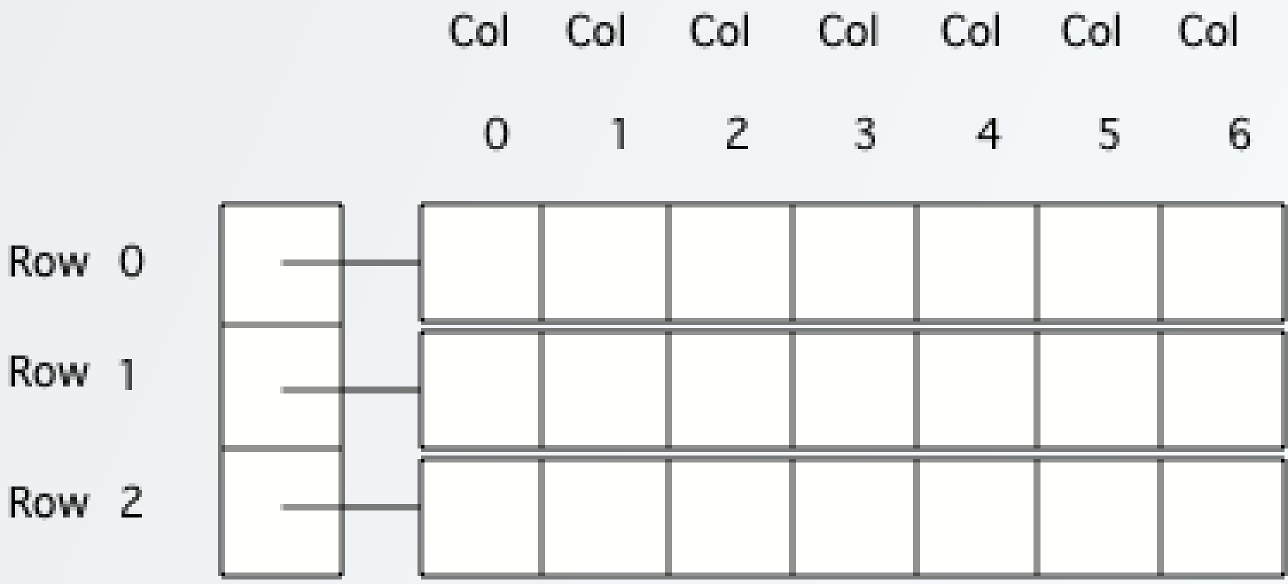
```
int[][] my2DArray = new int[3][7];
```

Arrays of Arrays

Ultimately, a 2D array is an array of arrays.

```
int[][] my2DArray = new int[3][7];
```

creates a new array containing three `int[]`s. Each `int[]` (row) has a length of seven.



Types & Expressions for 2D arrays

If we declare

```
double[][] matrix = new double[100][30];
```

To reference a single element at row `r` and column `c`:

```
double value = matrix[r][c]; // "row-major order"
```

Expression	Type	Description
matrix	double[][]	2D array of doubles, or an array of double arrays
matrix[3]	double[]	Array of doubles
matrix[3][4]	double	double

Explicit 2D Array Initialization

Sometimes helpful to write out the 2D array explicitly in your program:

```
int[][] scores = {{44, 55, 66, 77},
                  {88, 99, 11, 22},
                  {33, 0, 100, 50}};
```

Check in:

- What is the type of `scores`?
- What is the type of `scores[1]`?
- What is the value of `scores[1][2]`?
- What is the index for the value `22`?

Explicit 2D Array Initialization

Sometimes helpful to write out the 2D array explicitly in your program:

```
int[][] scores = {{44, 55, 66, 77},
                  {88, 99, 11, 22},
                  {33, 0, 100, 50}};
```

Check in:

- What is the type of `scores`? **int[][]**
- What is the type of `scores[1]`? **int[]**
- What is the value of `scores[1][2]`? **11**
- What is the index for the value `22`? **scores[1][3]**

Shapes of 2D Arrays

A 2D array can have a different number of rows and columns.

```
String[][] square = new String[5][5];
String[][] rectangle = new String[5][10];
```

We say a 2D array is **square** if it has the same number of rows and columns. Otherwise, we say it is **rectangular** if all rows have the same length but the number of rows and columns are different.

Iterating over Squares & Rectangles

We can use nested for-loops to iterate over the elements of a 2D array.

```
double[][] randomRectangle = new double[3][10];
int numRows = randomRectangle.length;
int numCols = randomRectangle[0].length; // NPE if row 0 is null, so be careful!
for (int row = 0; row < numRows; row++) {
        for (int col = 0; col < numCols; col++) {
                randomRectangle[row][col] = Math.random();
        }
}
```

Initializes a rectangular array of doubles to store 30 random values.

Shapes of 2D Arrays

Since 2D arrays are *arrays of arrays*, there's no actual requirement that each "inner" array has the same length as all the others. That is, rows may have different numbers of columns.

```
String[][] names = {{"Harry"},
                    {"Becca", "Bhrajit"},
                    {"Adi", "Sukya", "Sofia"}};
```

`names` is an example of a **ragged** or **jagged** 2D array. (Both terms are used for the same thing)

Dangers of Jagged Arrays

Consider the following example:

```
// same 2d array as before, just on one line for space purposes
String[][] names = {{"Harry"}, {"Becca", "Bhrajit"}, {"Adi", "Sukya", "Sofia"}};
int numRows = names.length;
int numCols = names[0].length; // NPE if row 0 is null, so be careful!
for (int row = 0; row < numRows; row++) {
        for (int col = 0; col < numCols; col++) {
                System.out.println(names[row][col]);
        }
}
```

What gets printed?

Dangers of Jagged Arrays

Consider the following example:

```
// same 2d array as before, just on one line for space purposes
String[][] names = {{"Harry"}, {"Becca", "Bhrajit"}, {"Adi", "Sukya", "Sofia"}};
int numRows = names.length;
int numCols = names[0].length; // NPE if row 0 is null, so be careful!
for (int row = 0; row < numRows; row++) {
        for (int col = 0; col < numCols; col++) {
                System.out.println(names[row][col]);
        }
}
```

`numCols` is set to `1` and doesn't change without each iteration, so we print only the first String from each column.

Dangers of Jagged Arrays

Consider the following example:

```java
// same 2d array as before, just on one line for space purposes
String[][] names = {{"Adi", "Sukya", "Sofia"}, {"Harry"}, {"Becca", "Bhrajit"}};
int numRows = names.length;
int numCols = names[0].length; // NPE if row 0 is null, so be careful!
for (int row = 0; row < numRows; row++) {
        for (int col = 0; col < numCols; col++) {
                System.out.println(names[row][col]);
        }
}
```

What gets printed?

Dangers of Jagged Arrays

Consider the following example:

```java
// same 2d array as before, just on one line for space purposes
String[][] names = {{"Adi", "Sukya", "Sofia"}, {"Harry"}, {"Becca", "Bhrajit"}};
int numRows = names.length;
int numCols = names[0].length; // NPE if row 0 is null, so be careful!
for (int row = 0; row < numRows; row++) {
        for (int col = 0; col < numCols; col++) {
                System.out.println(names[row][col]);
        }
}
```

The program crashes! `names[1]` has only one element, so `names[1][0]` is valid, but `names[1][1]` is not.

Practice Safe Iteration

When iterating over a 2D array, always use the `length` of the inner array to determine the number of columns.

```
// same 2d array as before, just on one line for space purposes
for (int row = 0; row < names.length; row++) {
        for (int col = 0; col < names[row].length; col++) {
                System.out.println(names[row][col]);
        }
}
```

This works for any 2D array (as long as it's not `null` and none of its rows are `null`)

Practice Safest Iteration

When iterating over a 2D array, always use the `length` of the inner array to determine the number of columns. Also, check for `null` values

```
// same 2d array as before, just on one line for space purposes
for (int row = 0; names != null && row < names.length; row++) {
        for (int col = 0; names[row] != null && col < names[row].length; col++) {
                System.out.println(names[row][col]);
        }
}
```

This works for any 2D array at all, but is often overkill if you can be sure there are no null values ahead of time.