Arrays

COSC 1P03 – Lecture 01 (Spring 2024)

Maysara Al Jumaily

amaysara@brocku.ca

Brock University

Tuesday April 30, 2024

Total slides: 21

Lecture Outline

01 Intuition of Arrays

02 1-Dimensional Arrays

- Definition of 1-Dimensional Array
- Declaring/Initializing 1-Dimensional Array
- Populating and Traversing 1-Dimensional Array
- 1-Dimensional Array as a Parameter and Return type
- Right-Sized Array Versus Variable-Sized Array
- Initializing Arrays of Primitive Types and Objects
- Fact: Array Size Cannot be Changed

03 2-Dimensional Array

- Definition of 2-Dimensional Array
- Row-Major Traversal
- Column Major Traversal

04 Arrays in a Nutshell

05 Arrays in Memory

Intuition of Arrays

- Consider the following task:
  - Create a simple program that holds five `int`s
  - Now, increment each integer by 2
  - How about decrement each integer by 7 instead
  - Okay... Now, assume we had 500 elements instead of just five; changing the values manually isn't practical any more
- Since we are changing all the elements we have, there should be a way to use a `for` loop to iterate through all of them and change each element to the desired output
- Also we don't want to manually create 500 element
- How can we store a table in java? How can we make a spreadsheet similar in Excel?

Definition of 1-Dimensional Array

- A 1-D array is a type of data structure that hold multiple elements of the same type that can be accessed by an index
  - All elements must be the same type (*i.e.*, all must be `int`s or all must be `double`s, etc.)
- Indices are **zero-based**
- Here is an integer array, named `numbers`, which holds 5 `int` elements:

`numbers`	213	54	8	100	75
Index	0	1	2	3	4
Access Syntax	`numbers[0]`	`numbers[1]`	`numbers[2]`	`numbers[3]`	`numbers[4]`

Figure: An array named `numbers` of type `int` holding five `int` elements

- Arrays have a special property called `length` which returns the number of cells in the array as **one-based**
  - Printing `numbers.length` gives 5 not 4 (largest index ≠ length)
  - The `length` property is *not* a method (*don't* add parentheses at the end)
- Valid indices are between `0` and `numbers.length - 1` (including both boundaries), otherwise, `IndexOutOfBoundsException` is encountered
- Initially, Java will store the value `0` in `int`eger array cells

Declaring/Initializing 1-Dimensional Array I

- Here is a minimal but complete example showing how to declare and initialize an array as well as access the array's elements:

```
public class OneDimArray{
  private int[] numbers; //declared an array of ints
  public OneDimArray(){
    numbers = new int[5]; // initialized it to hold 5 cells
    System.out.println(numbers.length);//prints 5
    //stores  213, 54, 8, 100 and 75 in
    //indices  0 , 1 , 2,  3  and 4, respectively
    numbers[0] = 213; // store the value 213 at index 0
    numbers[1] = 54;  // store the value 54  at index 1
    numbers[2] = 8;   // store the value 8   at index 2
    numbers[3] = 100; // store the value 100 at index 3
    numbers[4] = 75;  // store the value 75  at index 4
  }
  public static void main(String[] args){
    OneDimArray oda = new OneDimArray();
  }
}
```

- The above is hardcoded values, which is not the norm in practical situations, but shows how arrays work

Declaring/Initializing 1-Dimensional Array II

- Another way to initialize an array is by *hardcoding* the values in it all at once (**tip**: use this when testing your program).
- Instead of accessing `number[0]`, `number[1]`, `number[2]`, `number[3]` and `number[4]` manually, we can do:

```
private void hardCodeValues(){
  int[] numbers;
  numbers = new int[]{213, 54, 8, 100, 75};
  // The above will have:
  // numbers[0] store 213
  // numbers[1] store 54
  // numbers[2] store 8
  // numbers[3] store 100
  // numbers[4] store 75
  System.out.println(numbers.length); // 5
}
```

- Or declare and initialize at once:

```
private void hardCodeValues(){
  int[] numbers = new int[]{213, 54, 8, 100, 75};
  System.out.println(numbers.length); // 5
}
```

Populating and Traversing 1-Dimensional Array

- Here is a method that will loop through each cell in an array and store the value 7 in all the elements. Lastly, it will print out the array:

```
private void populate(){
  int[] numbers; // declaring the array
  numbers = new int[10]; // initializing array to have 10 elements
  System.out.println(numbers.length); // prints 10
  // Valid indices are: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9
  for(int i = 0; i < numbers.length; i++){
    numbers[i] = 7;
  }
  for(int i = 0; i < numbers.length; i++){ // Print values on screen
    System.out.println(numbers[i]);
  }
}
```

- We cannot write `System.out.println(numbers);` to print the data
- When looping through *ALL* cells in the array, **never** have the condition to have an equal sign, it must be strictly less than:
  - `i <= numbers.length` ✗
  - `i < numbers.length` ✓
  - Remember: max valid index = `array.length - 1`

Passing 1-Dimensional Array as a Parameter

- Let us use the same method in the previous slide (without printing anything on screen) but now pass the array to populate as a parameter (with the assumption it has been initialized and holds 10 elements):

```
private void populate(int[] numbers){
  // Valid indices are: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9
  for(int i = 0; i < numbers.length; i++){
    numbers[i] = 7;
  }
}
```

Passing/Returning 1-Dimensional Array I

- Let us use the same method in the previous slide but now will return the same array and having all of the cells to hold the value 7:

```
private int[] getPopulatedArray(int[] numbers){
  // Valid indices are: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9
  for(int i = 0; i < numbers.length; i++){
    numbers[i] = 7;
  }
  return numbers;
}
```

- We can call this method and use its return type (we must pass an integer array):

```
int[] array = new int[10];
array = getPopulatedArray(array);
// right side of equal sign is evaluated first
```

- We can create two arrays and initialize one of them using the method

```
int[] arrayOne = new int[10]; // initialized normally
int[] arrayTwo = getPopulatedArray(arrayOne);
```

- We could also have a method that doesn't accept an array as a parameter but returns an array

Passing/Returning 1-Dimensional Array II

- Let us create a method that accepts the number of cells in the array along with the default value:

```
private int[] createArrayData(int lengthPassed, int initialValue){
  int[] result = new int[lengthPassed];
  for(int i = 0; i < result.length; i++){
    result[i] = initialValue;
  }
  return result;
}
```

- To use it, we could declare, initialize an array and assign it the returned array from the method:

```
int[] array = createArrayData(10, 7); // 10 cells storing value 7
```

- We can print it using a `for`-loop or `for`-each loop. The array is an `int`eger array, so each element is of type `int`:

```
for(int i = 0; i < array.length; i++){ // Using a for-loop
  System.out.println(array[i]);
}
for(int element: array){ // Using a for-each loop
  System.out.println(element);// No indices!
}
```

Right-Sized Array Versus Variable-Sized Array

- We need to differentiate between two concepts when creating an array:
- **Right-sized array:** We know exactly how many cells we need to have in our array, and all cells are a part of the data. This is known as *priori* or computable.
  - **Example:** We want to keep track of which months students were born in. We could create an array of size 12, where cell represents a month, and then increase the cell each time that month is encountered.
- **Variable-sized array:** We *don't* know how many cells to have, so we initialize the array to a large value/constant. Not all of the cells are a part of the data and size variable is needed.
  - **Example:** Recording the names of countries you will visit in your lifetime. We could say you will have a maximum of 200 cells, all initialized to the empty string `""` and each time you visit a country, you place its name in the array, and increment your size variable. Not all cells are a part of the data.

Initializing Arrays of Primitive Types and Objects

- The examples above used `int`eger arrays to convey the concepts
- We could use other primitive types `byte`s, `short`s, `long`s, `float`s, `double`s, `char`s and `boolean`s
- We could use objects as well, such as `String`s (yes, a `String` is an object), `Student`, `Item`, `Turtle`, etc.
- Here are some examples of declaring and initializing arrays of different types:

```
byte[] a = new byte[50];
short[] b = new short[26];
long[] c = new long[37];
float[] d = new float[4];
double[] e = new double[1];
char[] f = new char[1230];
boolean[] g = new boolean[783];

String[] h = new String[7];
Item[] i = new Item[7];
Turtle[] j = new Turtle[7];
Student[] k = new Student[7];
```

Fact: Array Size Cannot be Changed

- Once an array is initialized, its size **CANNOT** be changed, you cannot add nor remove from cells
  - You can re-initialize it, but this is not the same
  - Re-initializing will allow to change the size, correct, but also override the data stored in the array
- To correctly alter the size of an array:
  - Suppose you have the data stored in array called `numbers`
  - Create a new array, say, named `temp`, and initialize to the required size
  - Copy the values from the `numbers` array into the `temp` array
    - Loop through each cell in `numbers`
    - Copy the current value from `numbers`'s cell into `temp`'s cell
  - Re-initialize `numbers` to `temp`, *i.e.*, `numbers = temp;`
- The same concept goes to 2-D arrays, which are discussed next

Definition of 2-Dimensional Array

- A 2-D array is an array of arrays (*i.e.*, table) data structure that hold elements of the same type which are accessed by two indices `[r][c]`
  - All elements must be the same type (*i.e.*, all must be `int`s or all must be `double`s, etc.).
- Similar to 1-D arrays, 2-D arrays have indices that are **zero-based**
- The top-left cell is accessed through `array[0][0]`
- Here is a simple visual diagram of an array called `data` that holds 3 rows and 5 columns of `int`egers:

	Col 0	Col 1	Col 2	Col 3	Col 4	
`data[0]`	99 `data[0][0]`	72 `data[0][1]`	82 `data[0][2]`	56 `data[0][3]`	91 `data[0][4]`	Row 0
`data[1]`	61 `data[1][0]`	35 `data[1][1]`	62 `data[1][2]`	88 `data[1][3]`	27 `data[1][4]`	Row 1
`data[2]`	86 `data[2][0]`	82 `data[2][1]`	77 `data[2][2]`	83 `data[2][3]`	13 `data[2][4]`	Row 2

Figure: An array named `data` of type `int` holding 3 rows and 5 columns

- In a 2-D array, `data.length` returns the number of **rows** in the array and is **one-based**; In our example above, it returns the value 3

2-Dimensional Array II

- To declare a 2-D array of `int`s:
  ```
  private int[][] data; // instance variable of a 2-D array
  ```
- To initialize it so that is holds 3 rows and 5 columns:
  ```
  data = new int[3][5]; // [num of rows][num of columns]
  ```
- `System.out.println(data.length);` prints 3 (# of rows or 'height')
- To get the rth row, use `data[r]`, which is a 1-D array
- To get the cell at the cth row and cth column, use `data[r][c]`, which gets the `int`eger value stored in that cell
- For example, from the figure in the previous slide:
  - To get the value of the cell in row 1 and column 3, write the following to print 88:
    ```
    System.out.println(data[1][3]);
    ```
  - To set/update the value of the cell in row 1 and column 0 to the value 49, use the following:
    ```
    data[1][0] = 44;
    ```
- We can initialize and declare at once:
  ```
  int[][] data = new int[3][5];
  ```
- Of course, other types could also be used:
  ```
  String[][] names = new String[600][800];
  ```

2-Dimensional Array III

- To populate the 2-D array shown in the previous slide:

```
int[][] data = new int[3][5]; // [rows][columns]
data[0][0] = 99; // row 0 column 0
data[0][1] = 72; // row 0 column 1
data[0][2] = 82; // row 0 column 2
data[0][3] = 56; // row 0 column 3
data[0][4] = 91; // row 0 column 4
data[1][0] = 61; // row 1 column 0
data[1][1] = 35; // row 1 column 1
data[1][2] = 62; // row 1 column 2
data[1][3] = 88; // row 1 column 3
data[1][4] = 27; // row 1 column 4
data[2][0] = 86; // row 2 column 0
data[2][1] = 82; // row 2 column 1
data[2][2] = 77; // row 2 column 2
data[2][3] = 83; // row 2 column 3
data[2][4] = 13; // row 2 column 4
```

- This is for testing purposes only! Don't use this approach on homework/test! It is here to show the indices and the corresponding values (you will have to probably read the data from a file)

2-Dimensional Array IV

- Here is another approach to populate the same array as the previous slide:

```
int[][] data = new int[][]{
  {99, 72, 82, 56, 91}, // row 0
  {61, 35, 62, 88, 27}, // row 1
  {86, 82, 77, 83, 13}  // row 2
};
```

- This is for testing purposes only! Don't use this approach on homework/test! It is here to show the indices and the corresponding values (you will have to probably read the data from a file)

2-Dimensional Arrays – Row Major Traversal

- To loop through a 2-D array, *row-by-row* and print it out:

```
for(int r = 0; r < data.length; r++){ // rows
  for(int c = 0; c < data[r].length; c++){ // columns
    System.out.print(data[r][c] + ", ");
  }
  System.out.println(""); // new line to print the next row below it
}
```

- Now, using `ASCIIDisplayer` instead of `System.out.println(...);`:

```
for(int r = 0; r < data.length; r++){ // rows
  for(int c = 0; c < data[r].length; c++){ // columns
    display.writeInt(data[r][c]);
    display.writeString(", ");
  }
  display.newLine(); // new line to print the next row below it
}
```

- The meaning of `data[r].length` is: get the length of the rth row
- Oh, but all rows have the same length, so we can hardcode to 5 or just always get the length of the zeroth row (*i.e.*, `data[0].length`). That would work but *only* for rectangular/square arrays, not *Ragged/Jagged Arrays*

2-Dimensional Arrays – Column Major Traversal

- To loop through a 2-D array, *column-by-column* and print it out:

```
for(int c = 0; c < data[0].length; c++){ // columns
  for(int r = 0; r < data.length; r++){  // rows
    System.out.print(data[r][c] + ", ");
  }
  System.out.println(""); // new line to print the next row below it
}
```

- Now, using `ASCIIDisplayer` instead of `System.out.println(...);`:

```
for(int c = 0; c < data[0].length; c++){ // columns
  for(int r = 0; r < data.length; r++){  // rows
    display.writeInt(data[r][c]);
    display.writeString(", ");
  }
  display.newLine(); // new line to print the next row below it
}
```

Arrays in a Nutshell

- Arrays are a data structure that allows us to store multiple values of the same type in a variable.
- Arrays has the length property that returns the number of cells we have. It is **one-based**.
  - In a 1-D array, `array.length` returns the number of *columns* or 'width'. One for-loop is needed with `r < array.length` to loop through all columns
  - In a 2-D array, `array.length` returns the number of *rows* or "height". Two for-loops are needed with `r < array.length` (outer loop) to loop through all rows and `c < array[r].length` (inner loop) to loop through the columns in the rth row
    - This is row-major traversal, which is used more often than column-major traversal
- Remember, the indices of the cells are always 0-based. It means that your loops must always start at `0` and goes to `<` the length

Arrays in Memory

- A computer memory is a collection of cells associated with addresses (*i.e.*, numbered cells)
- Each primitive type/object has a size associated with it
- An `int` has a size of 4 bytes and a `long` has a size of 8 bytes, we denote the size as uppercase S
- Suppose our 1-D or 2-D array have zero-based indexing (which is the case in Java) The array starts at some zero-based memory address denoted with the letter a
- In 1-Dimensional arrays:
  - The memory address of the ith cell in the array is $a + (i \cdot S)$, where S is the size of element type in bytes
- In 2-Dimensional arrays (lexicographic (row-major) ordering):
  - The memory address of the ith row in the array is $a + (i \cdot T)$, where T is size of one row, which is $T = a[0].\text{length} \cdot S$
  - The address of some cell (i, j) is given as $a + (i \cdot S) + (j \cdot T)$
  - An example is the cell `a[i][j]` of an `int`eger array has its content in address $a + (i \cdot 4) + (j \cdot T)$
  - We have the size of an `int`eger type to be 4 bytes