Revised: 4/21/2020

# Arrays

Michael C. Hackett

Assistant Professor, Computer Science

Community College of Philadelphia

# Lecture Topics

- Arrays
  - Declaration/Initialization
  - Retrieving/Changing values of elements
- Iterating Through an Array
  - C-Style Loops
  - For-each Loops
- Copying
- Resizing
- Testing Equality
- The Linear Search Algorithm
- Multidimensional Arrays

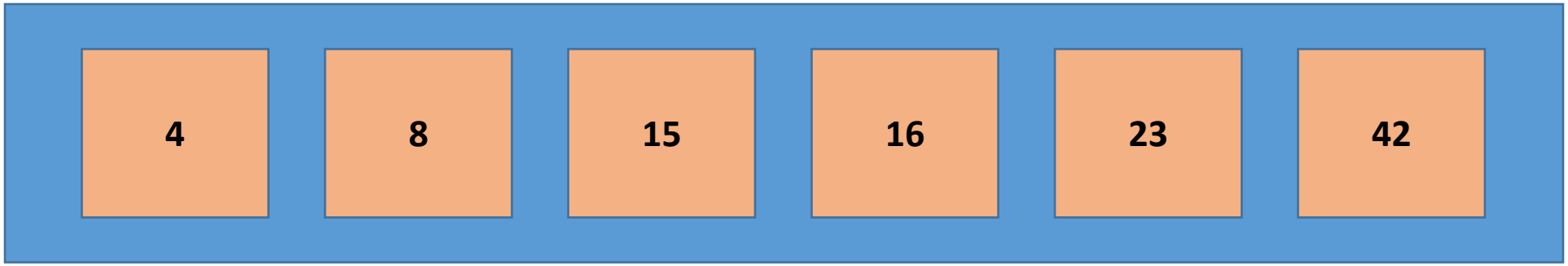
# Colors/Fonts

- Local Variable Names – Brown
- Primitive data types – Fuchsia
- Literals – Blue
- Keywords – Orange
- Object names – Green
- Operators/Punctuation – Black
- Field Names – Lt Blue
- Method Names – Purple
- Parameter Names – Gold
- Comments – Gray
- Package Names – Pink

Source Code – **Consolas**
Output – Courier New

# What is an array?

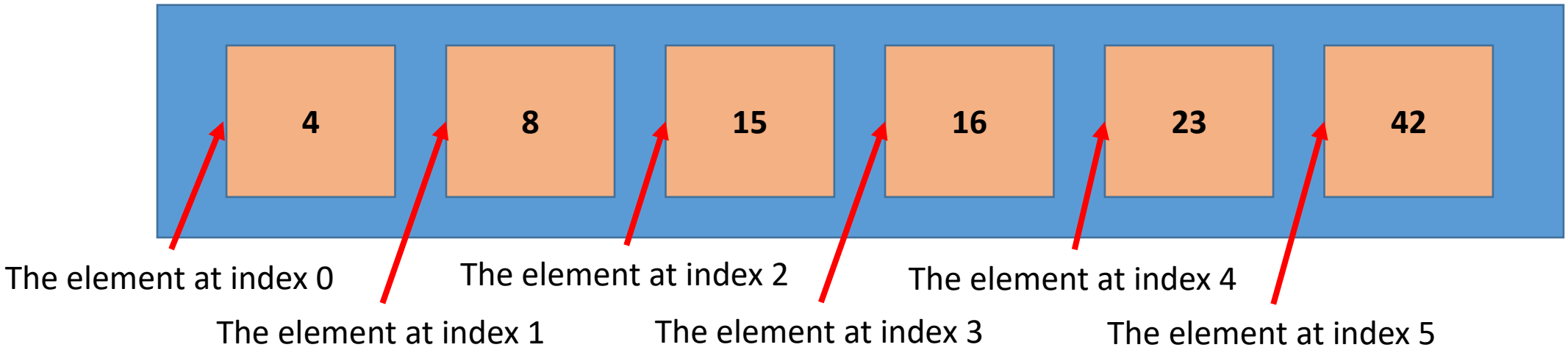
- An ***array*** is a container object that holds a fixed number of values.
- All values must be of the same data type or object (all ints, Strings, etc.)




An array of ints

# Array Terminology

- An ***index*** (or ***subscript***) is the number representing the position of an array element.
  - First index is always zero.
  - The index is always an int.
- An ***element*** is the data or object referenced by an index.



# Array vs List

- A List (like Python's List type) can:
  - Grow and shrink in size automatically.
    - Has functions that can insert, delete, or append data.
  - Heterogeneous: Not limited to containing one data type.
    - Can contain a mix of ints, doubles, strings, etc.

- Arrays are:
  - Fixed in length.
    - No functions to insert, delete, or append data.
  - Homogeneous: Limited to containing data of the same type.

# Declaring an Array

- Arrays are declared just like any other variable with one difference:
  - An open and close bracket is included after the data type

```
int[] numbers;

double[] values;

String[] names;
```

# Initializing an Array

- There are two ways to initialize an array:
  - Without data:

```
numbers = new int[6];
```

Data Type

- Has space for 6 ints.
- Default values are
  - 0 or 0.0 (for numeric arrays)
  - ' ' (for char)
  - false (for boolean)
  - null (for objects)

  - Or with data:

```
numbers = {4, 8, 15, 16, 23, 42};
```

Values are comma separated in open/close braces

# Declare and Initialize an Array

- Without data:

```
int[] numbers = new int[6];
```

- Or with data:

```
int[] numbers = {4, 8, 15, 16, 23, 42};
```

# Array Length

- The number of elements an array contains is referred to as the array's ***length***.

```
int[] numbers1 = new int[10];
int[] numbers2 = {5, 16, 12, 32, 41, 98};
```

- The numbers1 array has a length of 10 (indexes 0-9) and the numbers2 array has a length of 6 (indexes 0-5).
- An array's length **cannot** be changed after initialization.
  - We will later see a technique to "resize" an array.

# Array Length

- To retrieve an array's length, call on it's length field.
  - The length field is an int.

```
int[] numbers = new int[10];
int numbersLength = numbers.length;
System.out.println(numbersLength);
```

```
10
```

# Length limits to an array

- Arrays, technically, do not have a size limit, however…
  - Array indexes are represented by an int.
  - The maximum value of an int is 2,147,483,647.
  - Therefore, that is the maximum length of an array.

# Arrays and Memory

- Variables map to locations in memory using a ***symbol table***.
  - A symbol could be a variable or class or method, for example.
  - Managed by the operating system.

- A memory map is a diagram of memory addresses and the data associated with an address or addresses.
  - An address typically corresponds to 8 bits/1 byte of space.

# Arrays and Memory

Variables

```
int number = 58;
double value = 16.5;
char letter = 'L';
```

Symbol Table

| Symbol | Address |
|---|---|
| number | 1000 |
| value | 1004 |
| letter | 100C |

ints are 32 bits/4 bytes
doubles are 64 bits/8 bytes
chars are 16 bits/2 bytes

Memory Map

| Address | Data |
|---|---|
| 1000 | 58 |
| 1001 | |
| 1002 | |
| 1003 | |
| 1004 | 16.5 |
| 1005 | |
| 1006 | |
| 1007 | |
| 1008 | |
| 1009 | |
| 100A | |
| 100B | |
| 100C | L |
| 100D | |

# Arrays and Memory

Variables

```
char letter = 'L';
int[] numbers = {5, 3, 1}
```

Symbol Table

| Symbol | Address |
|---|---|
| letter | 1000 |
| numbers | 1002 |

Memory Map

| Address | Data |
|---|---|
| 1000 | L |
| 1001 | |
| 1002 | numbers[0] |
| 1003 | 5 |
| 1004 | |
| 1005 | |
| 1006 | numbers[1] |
| 1007 | 3 |
| 1008 | |
| 1009 | |
| 100A | numbers[2] |
| 100B | 1 |
| 100C | |
| 100D | |

# Arrays and Memory

- The OS only keeps track of the array's ***base address***.
  - The address where the array begins.

- The computer uses the following formula to calculate the address of other indexes in the array:

base address + index * byte size

# Arrays and Memory

Symbol Table

| Symbol | Address |
|---|---|
| `letter` | 1000 |
| `numbers` | 1002 |

Memory Map

| Address | Data |
|---|---|
| 1000 | L |
| 1001 | |
| 1002 | numbers[0] |
| 1003 | 5 |
| 1004 | |
| 1005 | |
| 1006 | numbers[1] |
| 1007 | 3 |
| 1008 | |
| 1009 | |
| 100A | numbers[2] |
| 100B | 1 |
| 100C | |
| 100D | |

numbers[0] = $1002_{16}$ + 0 * 4 = $1002_{16}$

numbers[1] = $1002_{16}$ + 1 * 4 = $1006_{16}$

numbers[2] = $1002_{16}$ + 2 * 4 = $100A_{16}$

- Memory addresses are represented using the hexadecimal system (Base-16).

# Initializing the Elements of an Array

- After initializing an empty array…

```
int[] numbers = new int[6];
```

- You can initialize the elements using the assignment operator and referencing the index using ***subscript notation***:

```
numbers[0] = 4;
numbers[1] = 8;
numbers[2] = 15;
numbers[3] = 16;
numbers[4] = 23;
numbers[5] = 42;
```

Assigns the value 4 to index 0

# Retrieving Elements from an Array

- To retrieve an array's element, again use subscript notation to reference the data at the particular index:

```
int[] multiplesOfTen = {10, 20, 30, 40, 50};

System.out.println(multiplesOfTen[0]);
System.out.println(multiplesOfTen[1]);
System.out.println(multiplesOfTen[2] + multiplesOfTen [3]);
```

```
10
20
70
```

# Changing the Values of Elements

- Assign new data to the array using subscript notation and the assignment operator.

```
int[] myTwoNumbers = new int[2];
myTwoNumbers[0] = 10;
myTwoNumbers[1] = 20;
System.out.println(myTwoNumbers[0]);
System.out.println(myTwoNumbers[1]);

myTwoNumbers[0] = 30;
myTwoNumbers[1] = 40;
System.out.println(myTwoNumbers[0]);
System.out.println(myTwoNumbers[1]);
```

Replaces 10 with 30

```
10
20
30
40
```

# ArrayIndexOutOfBoundsException

- This exception is caused by:
  - Trying to retrieve a value at an a non-existent index.
  - Trying to store a value to a non-existent index.
  - Using a negative as an index (`someArray[-1]`)

```
char[] letters = {'a', 'b', 'c'};
System.out.println(letters[3]);
```

```
run:
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
        at javaapplication3.JavaApplication3.main(JavaApplication3.java:23)
Java Result: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Iterating Through an Array

- For loops are ideal for iterating through the elements of an array.
  - The loop's counter can be used to represent each index.

```
String[] names = {"John", "Jane", "Jack"};

for(int i = 0; i < names.length; i++) {
   System.out.println(names[i]);
}
```

```
John
Jane
Jack
```

This loop will iterate from 0 through 2.

Will be 0, then 1, then 2

# Iterating Through an Array

- This for loop demonstrates the ability to initialize the elements of an array.

```
String[] names = new String[3];
Scanner keyboard = new Scanner(System.in);
for(int i = 0; i < names.length; i++) {
    System.out.print("Enter name #" + i + ": ");
    names[i] = keyboard.nextLine();
}

//Prints the values of the names array
for(int i = 0; i < names.length; i++) {
    System.out.println(names[i]);
}
```

```
Enter name #0: John
Enter name #1: Jane
Enter name #2: Jack
John
Jane
Jack
```

# Iterating Through an Array

- You may have noticed the output started by asking for name #0.
  - It would look better if it started by asking for name #1
  - We would still want to assign that name to index 0, though.

```
Enter name #0: John
Enter name #1: Jane
Enter name #2: Jack
John
Jane
Jack
```

# Iterating Through an Array

- This change will add one to i when printed.
  - But it won't actually replace the current value of i.

```
String[] names = new String[3];
Scanner keyboard = new Scanner(System.in);
for(int i = 0; i < names.length; i++) {
   System.out.print("Enter name #" + (i + 1) + ": ");
   names[i] = keyboard.nextLine();
}
```

```
Enter name #1: John
Enter name #2: Jane
Enter name #3: Jack
```

# Iterating Through an Array

- This for loop demonstrates the ability to change or alter the values of an array.

```
String[] names = {"John", "Jane", "Jack"};

for(int i = 0; i < names.length; i++) {
   names[i] = names[i].toUpperCase();
}
//Prints the values of the names array
for(int i = 0; i < names.length; i++) {
   System.out.println(names[i]);
}
```

Replaces the original value with an uppercase version of itself

```
JOHN
JANE
JACK
```

# Iterating Through an Array

- This for loop iterates through the elements backwards.

```
String[] names = {"John", "Jane", "Jack"};

for(int i = names.length - 1; i >= 0; i--) {
   System.out.println(names[i]);
}
```

Starts at the last index

Stops when -1 is reached

```
Jack
Jane
John
```

# Iterating Through an Array

- This for loop iterates through a portion of the array.

```
String[] names = {"John", "Jane", "Joe", "Jack"};

for(int i = 0; i < names.length/2; i++) {
   System.out.println(names[i]);
}
```

```
John
Jane
```

# For-Each Loop

- The ***for-each loop*** (also known as ***enhanced for loop*** or ***for-in loop***) is special type of for loop that iterates over the contents of an array or list.
  - This is the type of for loop Python uses.

```
for(dataType variableName : arrayName) {
   ...
}
```

- For each element in the array or list, *variableName* will represent that element for each iteration.
  - The data type of variableName must match the data type of the array.

# For-Each Loop

```
String[] names = {"John", "Jane", "Joe", "Jack"};

for(String name : names) {
   System.out.println(name);
}
```

```
John
Jane
Joe
Jack
```

# For-Each Loop

- For-each loops will iterate over the entire length of the array.
  - Even if there is no element present.

```
String[] names = new String[3];
names[0] = "John";
names[1] = "Jane";

for(String name : names) {
   System.out.println(name);
}
```

```
John
Jane
null
```

No String stored at index 2

# For-Each Loop Capabilities

- It is the preferred way to iterate over every element, from start to finish.
- No need to worry about array size/length.
- No need to worry about any ArrayIndexOutOfBoundsExceptions.

# For-Each Loop Limitations

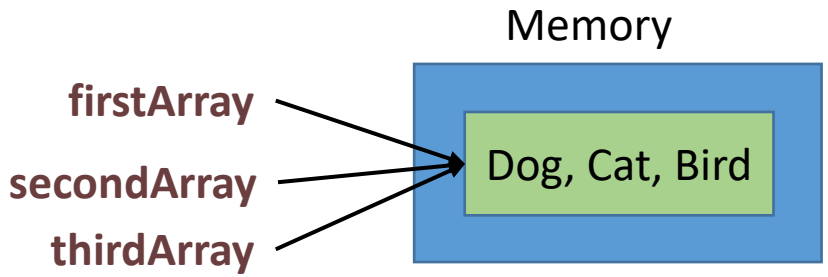
- Can’t change the elements in the array.
- Can’t go in reverse.
- Can’t iterate over a portion of the array.
- Can’t work with additional arrays in the loop.
  - For example, copying elements from one array to another.
- Doesn’t keep track of subscripts/index numbers.
  - There’s no counter variable like a traditional for loop.

# Copying Arrays

- Copying an array like the example below creates a ***shallow copy.***
  - Shallow copies are multiple variables referencing the same data.

```
String[] firstArray = {"Dog", "Cat", "Bird"};
String[] secondArray = new String[5];

secondArray = firstArray;
String[] thirdArray = firstArray;
```



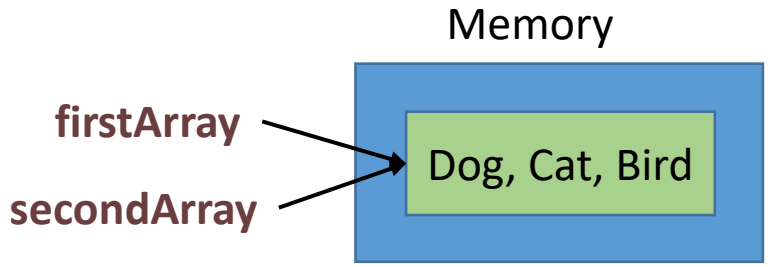
# Shallow Copies

- Since the variables reference the same array, changing one appears to change any others.

```
String[] firstArray = {"Dog", "Cat", "Bird"};
String[] secondArray = firstArray;
System.out.println(firstArray[0]);

secondArray[0]= "Fish";
System.out.println(firstArray[0]);
```

```
Dog
Fish
```



# Copying Arrays

- To create a second, separate array with the same contents you need to perform a ***deep copy***.
  - A deep copy copies the contents of one array into a second array of the same length.

```
int[] original = {3, 5, 7, 9};
int[] copy = new int[original.length];

for(int i = 0; i < original.length; i++) {
   copy[i] = original[i];
}
```

Same size

Copies the value to the corresponding index in the other array.

# Deep Copies

- Since the variables reference different arrays, changing one does not alter the original.

```
int[] original = {3, 5, 7, 9};
int[] copy = new int[original.length];

for(int i = 0; i < original.length; i++) {
    copy[i] = original[i];
}

System.out.println(original[0]);
copy[0]= 99;
System.out.println(original[0]);
```

```
3
3
```

# Increasing the Size of an Array

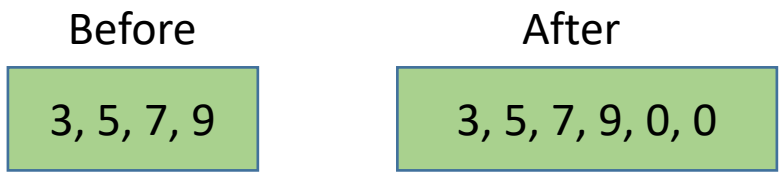
- To expand the length of an array:
  1. Create a second, temporary array with a longer length than the original.
  2. Deep copy the contents of the shorter array into the temporary array.
  3. Shallow copy the temporary array to the original's variable.
     - This will replace the original array, with the new bigger array.
  4. Set the temporary variable to null.
     - The variable no longer needs to reference the array.

# Increasing the Size of an Array

```
    int[] original = {3, 5, 7, 9};
1   int[] temporary = new int[original.length + 2];

2   for(int i = 0; i < original.length; i++) {
       temporary[i] = original[i];
    }

3   original = temporary;
4   temporary = null;
```




When making an array larger, new indexes are given the following default values:

- 0 (number type arrays)
- ' ' (char type arrays)
- false (boolean type arrays)
- null (object arrays)

# Decreasing the Size of an Array

- To shrink the length of an array:
  1. Create a second, temporary array with a shorter length than the original.
  2. Deep copy the contents of the longer array into the temporary array.
     - Not all will fit.
  3. Shallow copy the temporary array to the original's variable.
     - This will replace the original array, with the new smaller array.
  4. Set the temporary variable to null.
     - The variable no longer needs to reference the array.

# Decreasing the Size of an Array

```
    int[] original = {3, 5, 7, 9};
1   int[] temporary = new int[original.length - 2];

    for(int i = 0; i < temporary.length; i++) {
2      temporary[i] = original[i];
    }

3   original = temporary;
4   temporary = null;
```

# Testing Equality of Arrays

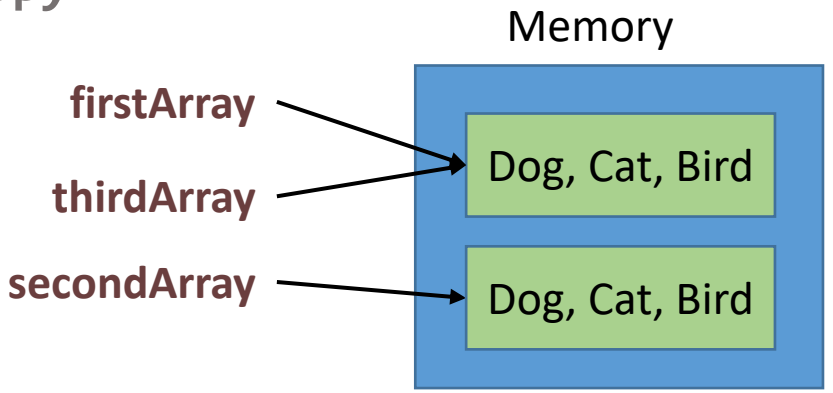
- Using the equality operator (==) to compare arrays only tests if the *reference* is equal, not the values/data.
  - In other words, == only tests if the two array variables are shallow copies.

```
String[] firstArray = {"Dog", "Cat", "Bird"};
String[] secondArray = {"Dog", "Cat", "Bird"};
String[] thirdArray = firstArray; //Shallow Copy

                  true
if(firstArray == thirdArray) {

}
                  false
if(firstArray == secondArray) {

}
```



# Testing Equality of Arrays

- Comparing equality of two arrays is normally done with a one-to-one comparison.
  - Index 0 of both arrays match, index 1 of both arrays match, and so on.

```
int[] firstArray = {3, 5, 7, 9};
int[] secondArray = {3, 5, 7, 9};

boolean equal = true;

for(int i = 0; i < firstArray.length; i++) {
    if(firstArray[i] != secondArray[i]) {
        equal = false;
        break;
    }
}
```

# Testing Equality of Arrays

- Two arrays are typically not equal if they don't have the same number of elements.
  - Checking they have equal lengths will also prevent an ArrayIndexOutOfBoundsException.

```
int[] firstArray = {3, 5, 7, 9};
int[] secondArray = {3, 5, 7};
boolean equal = true;
if(firstArray.length == secondArray.length) {
    for(int i = 0; i < firstArray.length; i++) {
        if(firstArray[i] != secondArray[i]) {
            equal = false;
            break;
        }
    }
}
else {
    equal = false;
}
```

# Linear Search (Sequential Search)

- A ***search algorithm*** is a series of steps that, when followed, tries to locate and/or retrieve information a set of data (ie. arrays).

- A linear search begins searching at the beginning of an array (index 0) and continuing until the item is found.

- Check index 0; if the element is not what you are looking for, continue to index 1; if the element is not what you are looking for, continue to index 2 (and so on…)

# Linear Search

- Checking to see if an array of ints contains the number 50.

```
int foundIndex = -1;

for(int i = 0; i<array.length; i++) {
   if(array[i] == 50) {
      foundIndex = i;
      break;
   }
}
```

Since we found what we needed, we can exit the loop.

# Linear Search

- Order of the elements (alphabetical, numerical, etc.) does not effect searching.
- Best case scenario: The information sought is the first element.
- Worst case scenario: The information sought is the last element.

# Multidimensional Arrays

- When an array contains arrays, it is called ***multidimensional***.
  - A one dimensional array:

```
int[] my1DArray = {2, 4, 6};
```

  - A two dimensional array:

```
int[][] my2DArray = {{8, 3, 7}, {1, 9, 9}, {5, 6, 9}};
```

# Multidimensional Arrays

- It's often better to write two dimensional arrays like this:

```
int[][] my2DArray = {{8, 3, 7},
                     {1, 9, 9},
                     {5, 6, 9}};
```

- This way, it's easier to see each "row" (first dimension) and "column" (second dimension).

# Multidimensional Arrays

- Empty two dimensional arrays are initialized by specifying the number of rows (first) and columns (second):

```
int[][] my2DArray = new int[3][4];
```

# Multidimensional Arrays

- Elements in a two dimensional array are referenced by row and column:
  - Row and column numbers start at zero.

```
int[][] my2DArray = {{8, 3, 7},
                     {1, 9, 9},
                     {5, 6, 9}};

my2DArray[1][2] = 2; //Assignment
System.out.println(my2DArray[0][1]); //Retrieval/Prints 3
```

# Multidimensional Arrays

```
int[][] my2DArray = {{2, 4, 6},
                     {1, 3, 5},
                     {3, 6, 9},
                     {1, 2, 3}};
```

What element is at `my2DArray[0][2]`?

What element is at `my2DArray[3][1]`?

What element is at `my2DArray[1][0]`?

# Multidimensional Arrays

- Rows in a multidimensional array do not have to be the same length.
  - This is called a ***Ragged Array***.

```
int[][] my2DArray = {{2, 4, 6},
                     {1, 3},
                     {9},
                     {1, 2, 3, 4}};
```

- Be careful with ragged arrays as not all rows have the same number of columns.

`my2DArray[2][1]` does not exist, even though every other row has a column 1.

# Multidimensional Arrays

- Two for loops are required to iterate through a two dimensional array.

```
int[][] my2DArray = {{8, 3},
                     {1, 9}};

for(int i = 0; i < my2DArray.length; i++) {
   for(int j = 0; j < my2DArray[i].length; j++) {
       System.out.println(my2DArray[i][j]);
   }
}
```

Rows

Columns

# Multidimensional Arrays

- Iteration through a two dimensional array using enhanced for loops.

```
int[][] my2DArray = {{8, 3},
                     {1, 9}};

for(int[] row : my2DArray) {
   for(int col : row) {
       System.out.println(col);
   }
}
```

Rows

Columns

# Multidimensional Arrays

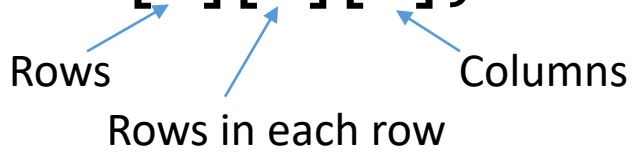
- There is no limit to the number of dimensions an array can have.

- A three dimensional array:

```
int[][][] my3DArray = {{{4,8},{15,16,23,42}},{{11,33},{22,44}}};
```

- In the case of a three dimensional array, the rows themselves have rows.

```
int[][][] my3DArray = new int[2][2][3];
```



# Multidimensional Arrays

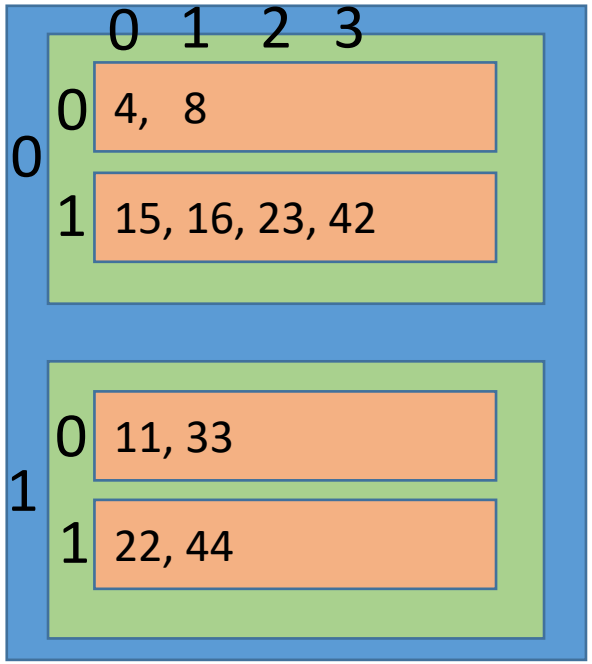
```
                                  0   1    2    3
int[][][] my3DArray = {{{4, 8},0
                        {15,16,23,42}},1
                       {{11,33}, 0
                        {22,44}}}; 1
```

0

1

What element is at `my3DArray[0][1][2]`?

What element is at `my3DArray[1][0][0]`?



# Multidimensional Arrays

- Three for loops are required to iterate through a three dimensional array.

```
int[][][] my3DArray = {{{4, 8},
                        {15,16,23,42}},
                       {{11,33},
                        {22,44}}};

for(int i = 0; i < my3DArray.length; i++) {
    for(int j = 0; j < my3DArray[i].length; j++) {
        for(int k = 0; k < my3DArray[i][j].length; k++) {
            System.out.println(my3DArray[i][j][k]);
        }
    }
}
```

Columns

Inner Rows

Outer Rows

# Multidimensional Arrays

- Iteration through a three dimensional array using enhanced for loops.

```
int[][][] my3DArray = {{{4, 8},
                        {15,16,23,42}},
                       {{11,33},
                        {22,44}}};

for(int[][] outerRow : my3DArray) {
    for(int[] innerRow : outerRow) {
        for(int column : innerRow) {
            System.out.println(column);
        }
    }
}
```

Columns

Inner Rows

Outer Rows