Revised: 11/23/2025

Arrays II

Michael C. Hackett

Associate Professor, Computer Science

COMMUNITY COLLEGE OF PHILADELPHIA

Lecture Topics

- Resizing an Array
- Testing Array Equality
- Multidimensional Arrays
- Arrays and Methods
  - Arrays as Method Arguments
  - Methods that return Arrays
  - Variable-Length Arguments
- ArrayLists

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

Boolean expression is false

Boolean expression is true

Resizing an Array

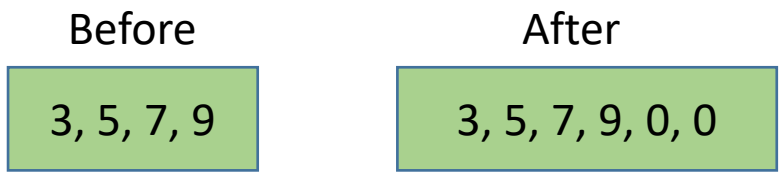
- To expand the length of an array:
  1. Create a second, temporary array with a longer length than the original.
  2. Deep copy the contents of the shorter array into the temporary array.
  3. Shallow copy the temporary array to the original's variable.
     - This will replace the original array, with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

Resizing an Array

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

    for(int i = 0; i < original.length; i++) {
2      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)

Resizing 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.

Resizing 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

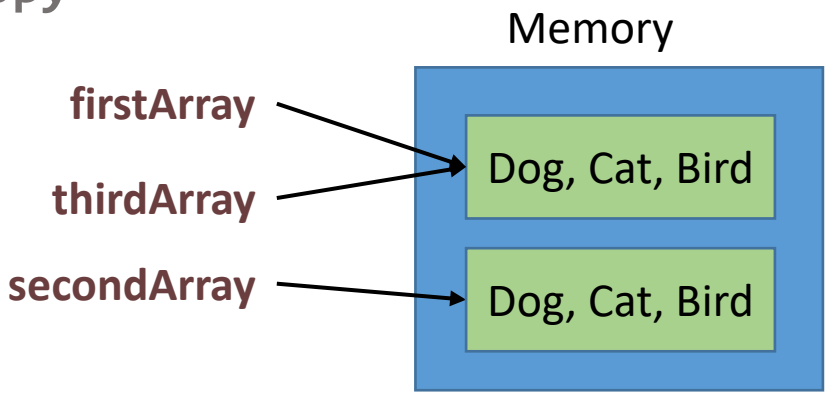
- Using the equality operator (==) to compare arrays only tests if the *reference* is equal, not the values/data.
  - In other words, == only tests if the two array variables are shallow copies.

```
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;
}
```

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

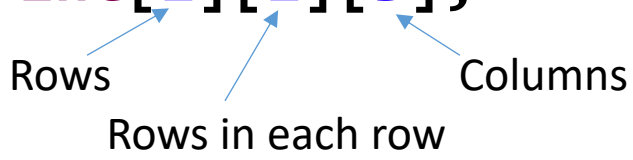
- There is no limit to the number of dimensions an array can have.

- 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

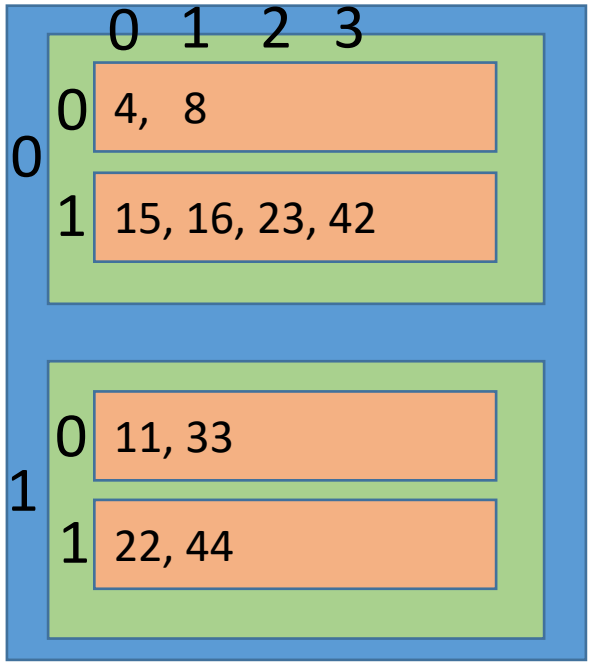
```
                                  0    1    2    3
int[][][] my3DArray = {{{4, 8},0
                        {15,16,23,42}},1
                       {{11,33}, 0
                        {22,44}}}; 1
```

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

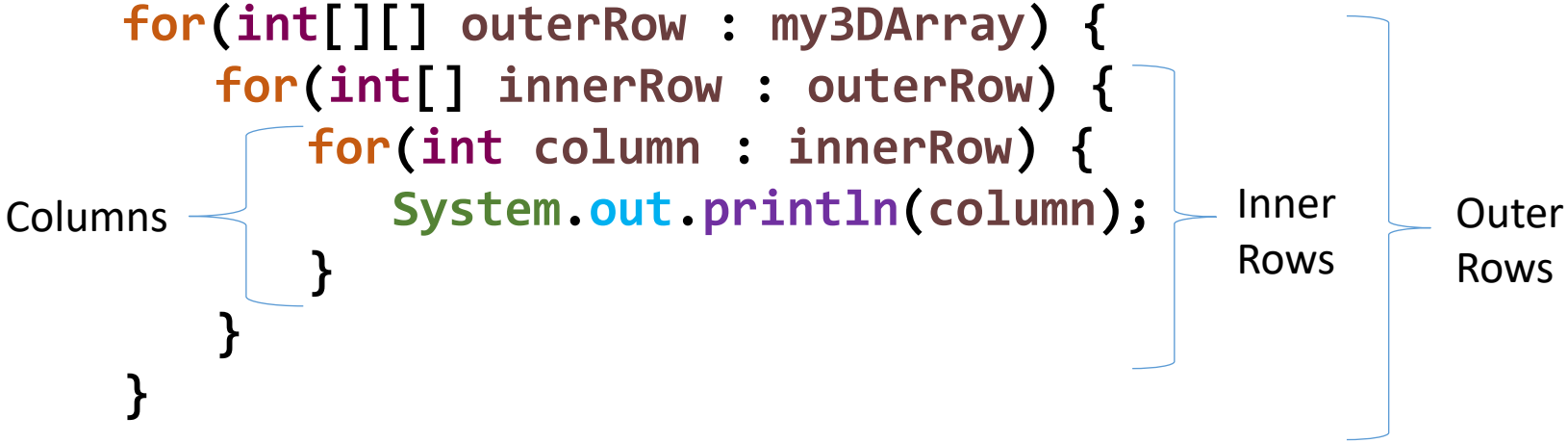
Outer Rows

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);
        }
    }
}
```



Arrays as Method Arguments

- An array can be passed to a method as an argument.

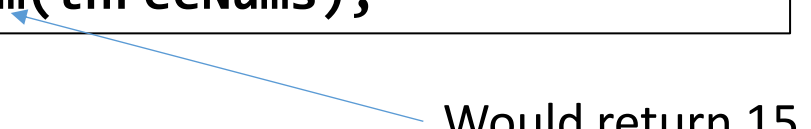
- Must match the array type specified as the parameter.

```
public int sum(int[] numbers)
```

Arrays as Method Arguments

```
public int sum(int[] numbers) {
    int sum = 0;
    for(int number : numbers) {
        sum += number;
    }
    return sum;
}
```

```
int[] threeNums = {4, 5, 6};
sum(threeNums);
```



Arrays as Method Arguments

- Arrays are always passed to a method **by reference**.

- **Pass by reference**- The reference to data is passed to the method.
  - Arrays and Objects are always passed by reference in Java.

- **Pass by value**- The data is passed to the method.
  - Primitive data are always passed by value in Java.

Passing by Value

```
public void demoMethod(int number) {
    number = 0;
}
```

Changes the number parameter, not x.

```
int x = 5;
demoMethod(x);
```

Passes x's value as the argument.

Passing by Reference

```
public void demoMethod(int[] array) {
    array[1]= 0;
}
```

Changes the threeNums array.

```
int[] threeNums = {4, 5, 6};
demoMethod(threeNums);
```

Passes threeNums's reference as the argument.

Variable Length Arguments

- ***Variable Length Arguments*** (or ***varargs***) allow a method to accept an undetermined number of parameters/arguments.

```
public int sum(int... numbers)
```

- The varargs must all be of the correct type.

- The varargs will be treated as an array inside the method.
  - Varargs *are* arrays, just not declared as such.

Variable Length Arguments

```
public int sum(int... numbers) {
    int sum = 0;
    for(int number : numbers) {
        sum += number;
    }
    return sum;
}
```

`sum(4, 5, 6);`	Valid. Would return 15.
`sum(2, 3);`	Valid. Would return 5.
`sum(7, 8.5);`	Not valid.

Variable Length Arguments

```
public int sum(int... numbers) {
    int sum = 0;
    for(int number : numbers) {
        sum += number;
    }
    return sum;
}
```

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

sum(myOriginalArray);
```

You can pass an array to a vararg. The sum method would return 24 in this example.

Variable Length Arguments

- No additional parameters can follow a vararg.

```
public int doMath(int... numbers, String operationType) {
```
INVALID

- Although, there can be any number of normal parameters preceding it.

```
public int doMath(String operationType, int... numbers) {
```
VALID

Variable Length Arguments

```
public int doMath(String operationType, int... numbers) {
    int answer = 0;
    if(operationType.equals("+")) {
        for(int number : numbers) {
            answer += number;
        }
    } else if(operationType.equals("*")) {
        answer = 1;
        for(int number : numbers) {
            answer *= number;
        }
    }
    return answer;
}
```

```
doMath("+", 4, 5, 6);
doMath("*", 7, 3);
```

Valid. Would return 15.
Valid. Would return 21.

Variable Length Arguments

```
public int doMath(String operationType, int... numbers) {
    int answer = 0;
    if(operationType.equals("+")) {
        for(int number : numbers) {
            answer += number;
        }
    } else if(operationType.equals("*")) {
        answer = 1;
        for(int number : numbers) {
            answer *= number;
        }
    }
    return answer;
}
```

```
int[] threeNums = {4, 5, 6};
doMath("+", threeNums);
```

Valid. Would return 15.

Returning an Array from a Method

- An array can be returned by a method.
  - Be sure the method's return type is an array.

```
public int[] getNumbers() {
   int[] threeNums = {4, 5, 6};
   return threeNums;
}
```

ArrayList

- A more sophisticated type of array.
- Size is changed as objects are added or removed.
- Must be imported: `import java.util.ArrayList;`

Adding to an ArrayList

- Use the add method to add a new element to the ArrayList.
  - One parameter – The data to be added.
- The item is inserted to the next available index.
  - In this example, that would be index 0 since the ArrayList is empty.

- The element you are adding must be of the correct type!

```
ArrayList<String> shoppingList = new ArrayList<String>();

shoppingList.add("Bread");
```

Getting an Element from an ArrayList

- Use the get method to retrieve an element from an ArrayList.
- One parameter – The index of the desired element.

```
ArrayList<String> shoppingList = new ArrayList<String>();

shoppingList.add("Bread");

System.out.println(shoppingList.get(0));
```

Output:

```
Bread
```

Inserting an Element into an ArrayList

- Use the add method to insert data to a particular index.
- Two parameters:
  - The index to insert at.
  - The data to add.
- This will **not** overwrite the data at that index

```
ArrayList<String> shoppingList = new ArrayList<String>();

shoppingList.add("Bread");
shoppingList.add("Milk");
shoppingList.add(1, "Eggs");

System.out.println(shoppingList.get(0));
System.out.println(shoppingList.get(1));
System.out.println(shoppingList.get(2));
```

Output:

```
Bread
Eggs
Milk
```

Removing an Element from an ArrayList

- Use the remove method.
- One parameter:
  - The index that you want to remove data from

```
ArrayList<String> shoppingList = new ArrayList<String>();

shoppingList.add("Bread");
shoppingList.add("Milk");
shoppingList.add(1, "Eggs");

System.out.println(shoppingList.get(0));
shoppingList.remove(0);
System.out.println(shoppingList.get(0));
```

Output:

```
Bread
Eggs
```

Getting the size of an ArrayList

- Use the size method to retrieve the number of elements in the ArrayList.

```
ArrayList<String> shoppingList = new ArrayList<String>();

shoppingList.add("Bread");
shoppingList.add("Milk");
shoppingList.add(1, "Eggs");

System.out.println(shoppingList.size());
```

Output:

3

Looping through an ArrayList

```
ArrayList<String> shoppingList = new ArrayList<String>();

shoppingList.add("Bread");
shoppingList.add("Milk");
shoppingList.add(1, "Eggs");

for(int i = 0; i < shoppingList.size(); i++) {
    System.out.println(shoppingList.get(i));
}
```

Output:

```
Bread
Eggs
Milk
```

Looping through an ArrayList (For each loop)

```
ArrayList<String> shoppingList = new ArrayList<String>();

shoppingList.add("Bread");
shoppingList.add("Milk");
shoppingList.add(1, "Eggs");

for(String item : shoppingList) {
    System.out.println(item);
}
```

Output:

```
Bread
Eggs
Milk
```

Checking if an ArrayList is empty

```
ArrayList<String> shoppingList = new ArrayList<String>();

shoppingList.add("Bread");
shoppingList.add("Milk");
shoppingList.add(1, "Eggs");

if(shoppingList.isEmpty()) {

}
```

Checking if an ArrayList contains an object

- The contains method returns a boolean
- Would return false in this example since the ArrayList does not have an element containing the String “Cheese”

```
ArrayList<String> shoppingList = new ArrayList<String>();

shoppingList.add("Bread");
shoppingList.add("Milk");
shoppingList.add(1, "Eggs");

if(shoppingList.contains("Cheese")) {

}
```

Getting the index of an object in an ArrayList

- The indexOf method returns an int:
  - The index of the element.
  - Will return -1 if the element is not in the ArrayList.

```
ArrayList<String> shoppingList = new ArrayList<String>();

shoppingList.add("Bread");
shoppingList.add("Milk");
shoppingList.add(1, "Eggs");

System.out.println(shoppingList.indexOf("Eggs"));
```

Output:

1

Getting the index of an object in an ArrayList

- “Eggs” is at both indexes 1 and 3.
- The indexOf method returns the index of the first occurrence.

```
ArrayList<String> shoppingList = new ArrayList<String>();

shoppingList.add("Bread");
shoppingList.add("Milk");
shoppingList.add(1, "Eggs");
shoppingList.add("Eggs");

System.out.println(shoppingList.indexOf("Eggs"));
```

Output:

```
1
```