# CMPS 200: Introduction to Programming Using JAVA

## LECTURE 9 – Arrays and ArrayList

Maurice J. KHABBAZ, Ph.D.

# Last Time

## Recursive Methods:

- Multiplication.
- Greatest Common Divisor.
- Factorial Computation.
- Fibonacci Sequence Generation.
- Towers of Hanoi.
- Palindromes.

## Small Bracket on Initializer Lists.

# Today

## Arrays:

- Definition (Reviewed).
- Array Declaration and Usage.
- Bounds Checking and Capacity.
- Arrays Storing Object References.
- Command Line Arguments.
- Arrays as Input/Output Method Parameters.
- Value and Reference Semantics.
- Variable Length Parameter Lists.
- Multidimensional Arrays.
- Introduction to Collections.

## Classes:

- `Arrays` class.
- `ArrayList` class.

# Arrays - Definition

- **Array:**
  - Simple but powerful construct used to group/organize data.
  - List of values having the same type.
  - Such a list is an **object** and is designated by a single name.
- **Array Element:**
  - A value stored at a specific positively numbered position in the array (*i.e*. an index).
    - Array element **indices** are **zero-based** (*i.e*. start at `0` and end at `# elements – 1`).
    - An array of size `N` is indexed from `0` to `N−1`.
  - An element type can be:
    - A primitive type (*i.e.,* `int`, `double`, `boolean`, `byte`, `short`, `long`, `float`, `char`)
    - An object reference (*e.g.,* `String`, `Integer`, `Double`, `Character`)
- **Example:** Writing a program to manage a list of 100 numbers:
  - Not practical to declare a separate variable for each of the numbers.
  - **Use an array:** declare one variable to hold multiple and individually accessible values.

# Array Declaration

- There are several ways to declare arrays.
- **Typical Syntax:** `<type>[] <a_name> = new <type>[<a_size>];`
- **Example:**

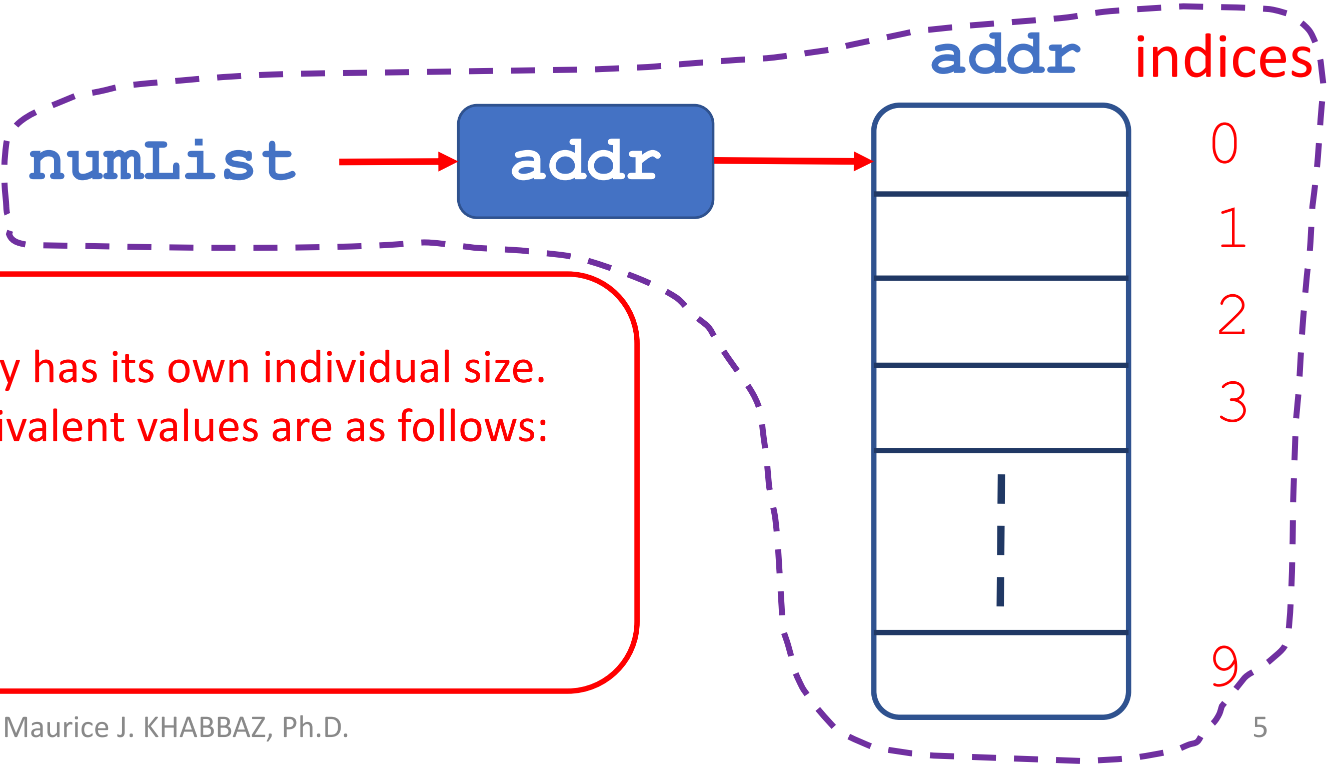
array size

```
int[] numList = new int[10];
```

an array of integers

array name

Memory Representation



- **Remarks:**
  1. The array type does not specify its size. Each array has its own individual size.
  2. By default, array elements are assigned zero-equivalent values are as follows:
     - `int → 0`
     - `double → 0.0`
     - `boolean → false`
     - `String → null`

# Array Declaration

- **Alternative Syntax:** Associate brackets with the array’s name instead of its type.

```
<type> <a_name>[] = new <type>[<a_size>];
```

- **Example:** The below two array declarations are the same:

```
int[] numList = new int[10];
int numList[] = new int[10];
```

always use this form as it is more readable and is consistent with other variable declarations.

# Array Declaration

- **Initializer Lists:**
  - Used to instantiate an array and associate its elements with initial values.
  - Values are delimited by curly braces `{ }` and separated by commas “`,`”.
  - When used, the `new` operator will no longer be required.
  - The size of the array will be determined by the number of elements in its initializer.
  - **Main Idea:** declare and initialize an array in one step.
  - **Syntax:** `<type>[] <a_name> = {<v1>, <v2>, ...};`
- **Example:**

```
               indices   0   1   2   3   4  5  6
int[] numList = {20, 31, 44, 88, 1, 0, 5};
```

# Accessing and Manipulating Array Elements

- A particular value in an array is referenced using:
  - The array name followed by the numerical index enclosed in between `[ ]`.
- **Example:**

```
int[] numList = {20, 31, 44, 88, 1, 0, 5};
System.out.println(numList[2]);
```

Output? **44**

`numList[2]` represents a single integer stored at a memory location and can be used wherever an integer variable can be used

```
numList[3] = -27;
```

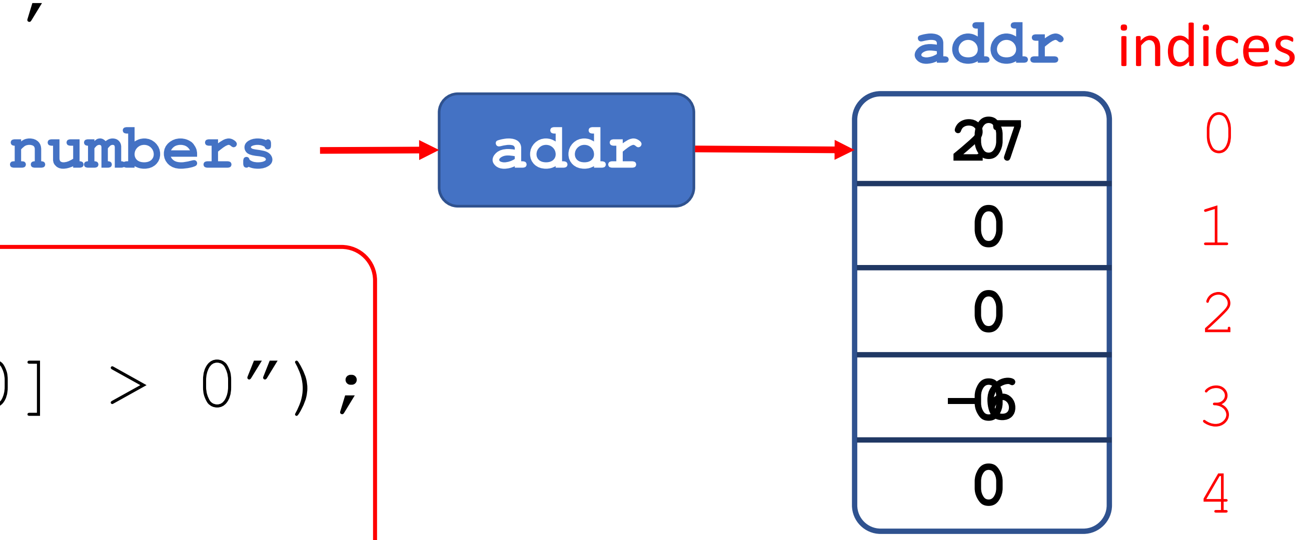
assigns the value `-27` to the fourth element in the array `numList`

```
int x = numList[0] / 2;
```

uses the first element of array `numList` in a computation.

# Another Example

```
int[] numbers = new int[5];
numbers[0] = 27;
numbers[3] = -6;
if (numbers[0] > 0)
   System.out.println(“n[0] > 0”);
else if (numbers[3] < 0)
   System.out.println(“n[3] < 0”);
else
   System.out.println(“Gotcha!”);
```



numbers → addr → addr

| addr | indices |
|---|---|
| 27 | 0 |
| 0 | 1 |
| 0 | 2 |
| -6 | 3 |
| 0 | 4 |

Output? **n[0] > 0**

# Exercise 1: Draw Array Structures

For each of the below, provide the corresponding array structure and indicate its values and indices.

```
double[] result = new double[5];
result[2] = 3.4;
result[4] = -0.5;
```

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| **result** | 0.0 | 0.0 | **3.4** | 0.0 | **-0.5** |

```
boolean[] tests = new boolean[6];
tests[3] = true;
```

| index | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **tests** | false | false | false | **true** | false | false |

# Checking Bounds

- Once an array is created, it has a fixed size, say, in general, `N`.
- An index used must specify a valid element:
  - Index value must be in the range `0` to `N-1`.
- **Example:**

```
int[] grades = new int[10];
/*
  Valid indices for grades are from 0 to 9.
  grades[10], grades[11], ... not allowed.
  Indices larger than 9 are out of bounds!
*/
```

# Automatic Bounds Checking (ABC)

- Whenever an array element is referenced, index used is checked:
  - `if (0 <= index && index < N)` → **VALID INDEX**.
  - `if (index < 0 || index >= N)` → **INVALID INDEX**.
- If an array index is out of bounds:
  - An `ArrayIndexOutOfBoundsException` (`AIOOBE`) is issued → **ABC**.
- **Example:** Assume having the array called `data`

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **data** | 15 | 20 | -1 | 5 | 0 | 1 | 10 | 6 | 9 | 11 |

| Reference | Outcome |
|---|---|
| `data[0]` | 15 |
| `data[9]` | 11 |
| `data[-1]` | AIOOBE |
| `data[10]` | AIOOBE |
| `data[data[5]]` | 20 |

# Off-By-One Errors

- An array’s indices begin at `0` and go up to `1` less than the array’s size.
- It is easy and common to fall into **off-by-one errors** in a program.
- **Example:**

```
double epsilon = 0.001;
double[] codes = new double[100];
for(int index = 0; index <= 100; index++)
     codes[index] = index * 50 + epsilon;
```

- **Problem:**
  - Above `for` loop processes all elements of codes and generates a run-time `AIOOBE`.

# Avoiding AIOOBE using the `length` variable

- An array has a **`public`** constant called **`length`** that stores its size.
- This `length` variable:
  - Holds the number of elements in an array (not the largest valid index)
  - **Reference Syntax:** `<a_name>.length` **[no need for `()` as for `String`]**
  - `<a_name>`'s elements' indices → `0` to `<a_name>.length-1`.

- **Example:** Print a list of stored integers in reverse order

```
int[] numbers = {4, 9, 5, 1, 0};
for(int i = numbers.length -1 ; i >= 0; i--)
    System.out.println(numbers[i]);
```

# Iterator Version of `for` Loop

- Appropriate only for processing absolutely all array elements in order:
  - From `0` to `<a_name>.length – 1`
- **Syntax:**

```
for(int <v> : <a_name>)
   <statment for processing <v>>
```

- **Example:**

```
int[] numbers = {4, 9, 5, 1, 0};
for(int n : numbers)
     System.out.println(n);
```

# Exercise 2: Array Mystery Problem

What is stored in array arr after the execution of the below JAVA code?

```
int[] arr = {1, 7, 5, 6, 4, 14, 11};
for (int i = 0; i < arr.length - 1; i++) {
   if (arr[i] > arr[i + 1]) {
        arr[i + 1] = arr[i + 1] * 2;
    }
}
```

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **arr** | 1 | 7 | 10 | 12 | 8 | 14 | 22 |

# Arrays of Objects

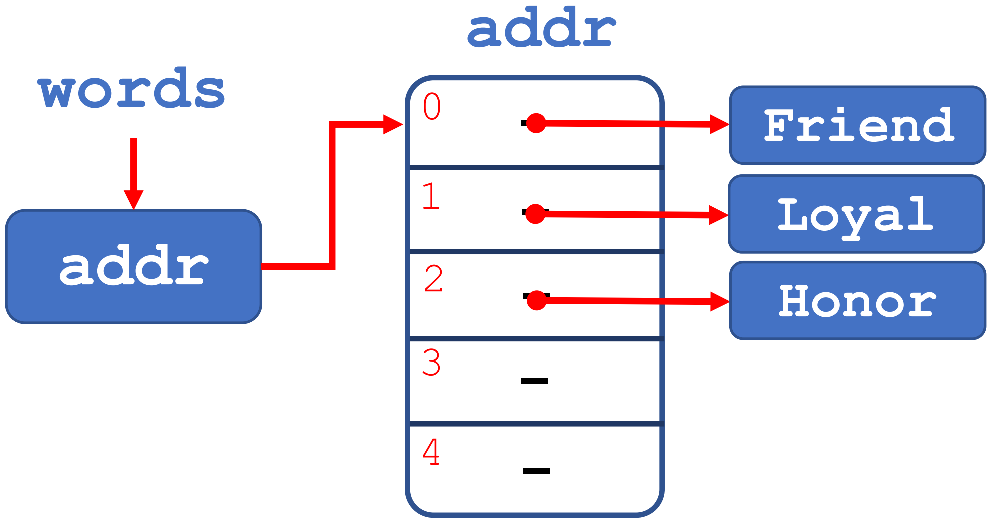
- The elements of an array can be references to non-scalar objects.
- **Example:**

```
String[] words = new String[5];
```

Reserves space to store five references to `String` objects

- Initially, an array of objects holds `null` ( – ) references.
- Each object stored in an array:
  - must be instantiated separately.
- **Example:**

```
words[0] = "Friend";
words[1] = "Loyal";
words[2] = "Honor";
```



# String Literals Revisited In Arrays Context

- String objects are created using string literals (*i.e.* “ . . . ”).

- **Example:** Create an array `verbs` and fill it with three `String` objects:

```
String[] verbs = {“play”, “study”, “sleep”};
```

# Command Line Arguments (CLAs)

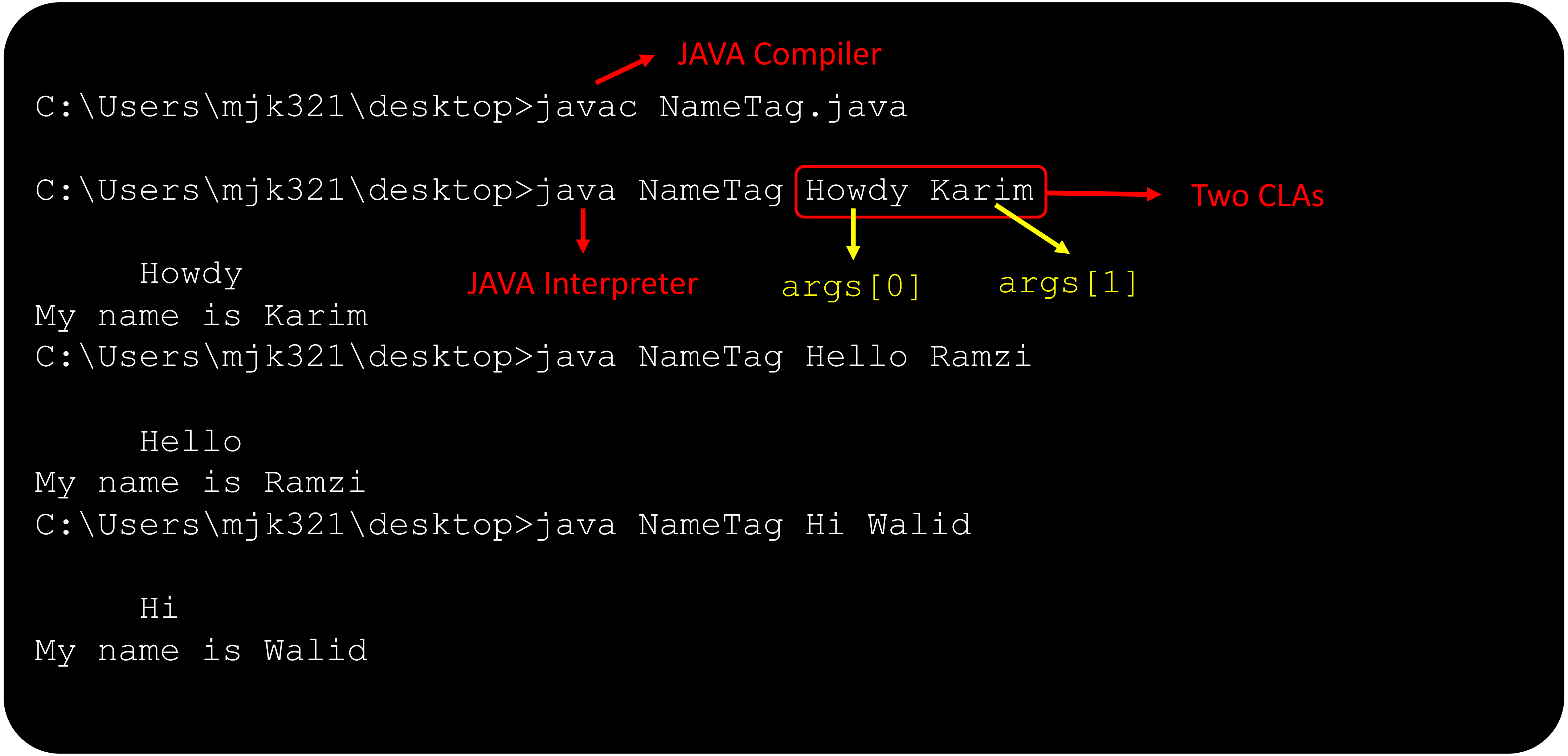
- `main()` method takes an array `args` of `String` objects as parameter.
- `args` (to mean arguments) represents the **command line arguments**:
  - Provided upon invoking the interpreter (`java.exe`) at the command prompt.
  - Separated by white space characters.
- **Example:**

If numeric data is passed to `main()` as CLA, appropriate processing is required to transform it from `String` to numeric type.

```
public class NameTag {
        public static void main(String[] args) {
                System.out.println();
                System.out.println(“          ” + args[0]);
                System.out.println(“My name is ” + args[1]);
        }
}
```

- Command Prompt interpreter invocation and execution of above code on next slide.

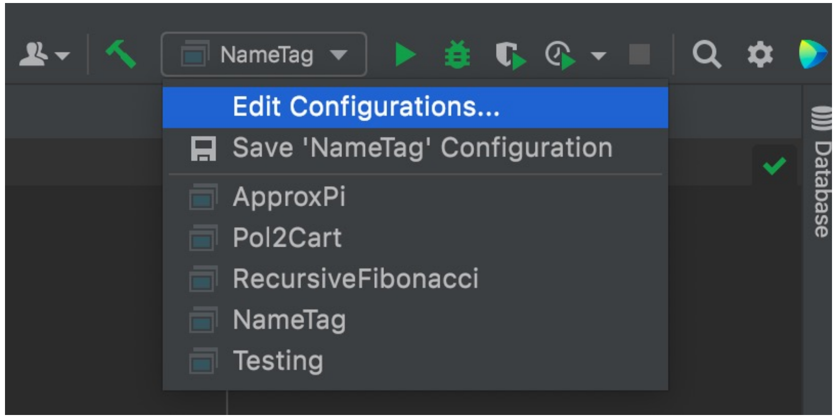
# Command Line Arguments (CLAs)



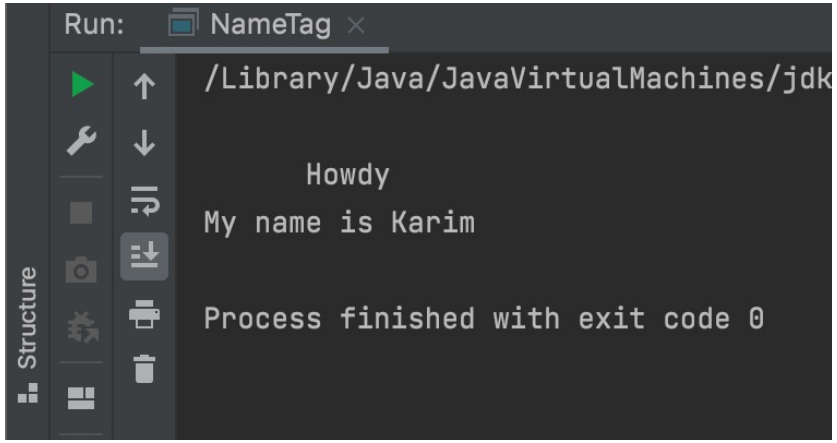
```
C:\Users\mjk321\desktop>javac NameTag.java

C:\Users\mjk321\desktop>java NameTag Howdy Karim

     Howdy
My name is Karim
C:\Users\mjk321\desktop>java NameTag Hello Ramzi

     Hello
My name is Ramzi
C:\Users\mjk321\desktop>java NameTag Hi Walid

     Hi
My name is Walid
```

# Command Line Arguments (CLAs) – IntelliJ IDEA

## Step 1

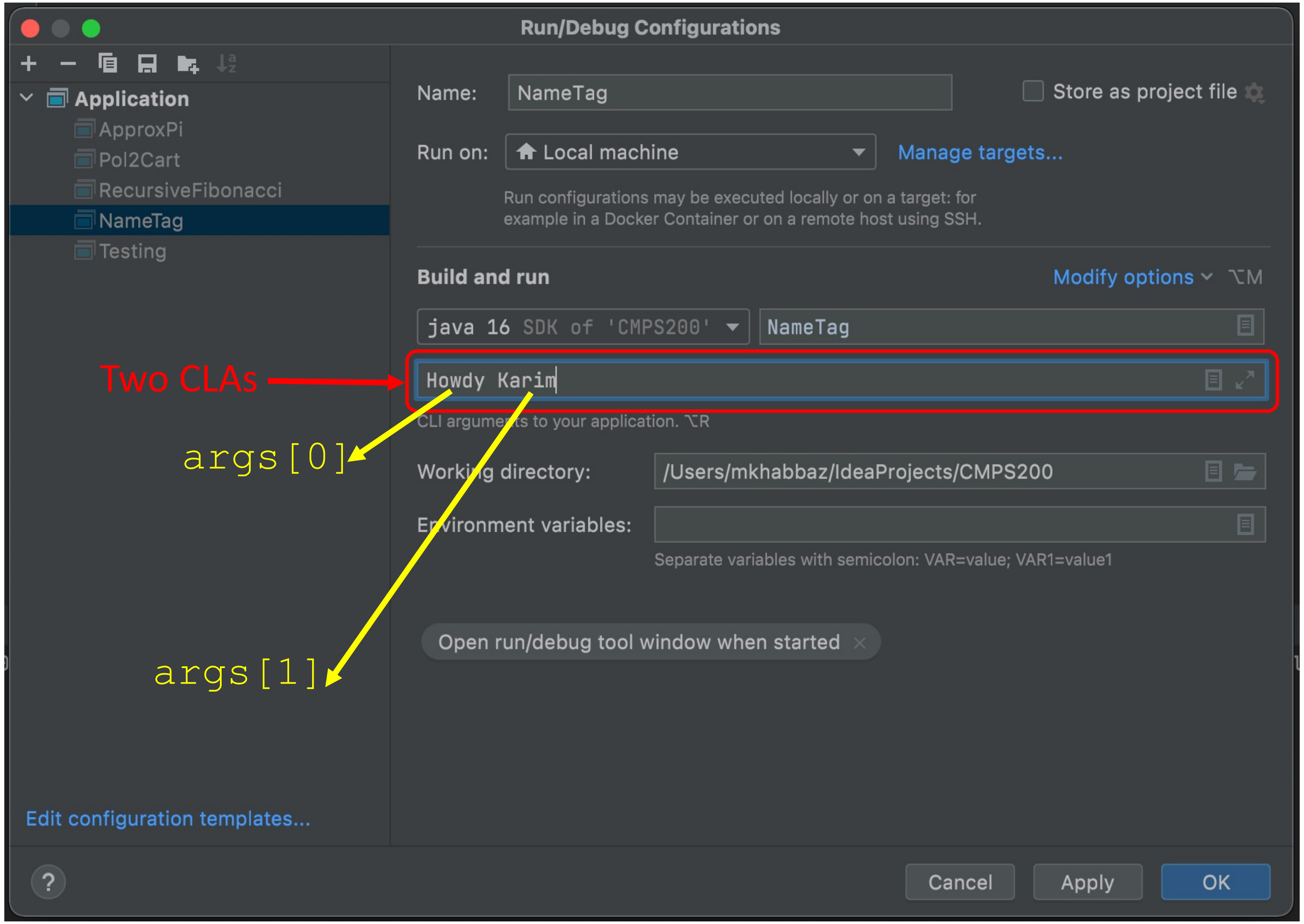


## Sample Execution



```
/Library/Java/JavaVirtualMachines/jdk

       Howdy
My name is Karim

Process finished with exit code 0
```

## Step 2



Name: NameTag

Run on: Local machine

Run configurations may be executed locally or on a target: for example in a Docker Container or on a remote host using SSH.

java 16 SDK of 'CMPS200'    NameTag

Two CLAs → `Howdy Karim`

CLI arguments to your application. ⌥R

args[0]

args[1]

Working directory: /Users/mkhabbaz/IdeaProjects/CMPS200

Environment variables:

Separate variables with semicolon: VAR=value; VAR1=value1

# Arrays As Input Parameters To Methods

- **Typical Header Syntax:**

```
public static <type> <methodName>(<a_type>[] <a_name>, ...)
```

- **Example:**

Write a method `average()` to return the mean of integers stored in an array.

```
public static double average(int[] integers) {
   int sum = 0;
   for (int i = 0; i < integers.length; i++)
      sum += integers[i];
   return (double) sum / integers.length;
}
```

More elegant to use an iterator:

```
for (int i : integers)
   sum += i;
```

- **Remark:**
  - `integers` is just an array having an unspecified arbitrary size. Yet, it can be processed.

# average(): Complete Example

```
public class ComputeAverage {
    public static double average(int[] integers) {
        int sum = 0;
        for (int i = 0; i < integers.length; i++)
            sum += integers[i];
        return (double) sum / integers.length;
    }
    public static void main(String[] args) {
        int[] grades = {34, 56, 84, 21, 13, 70};
        double avg = average(grades);
        System.out.println(“Exam Average: ” + avg);
    }
}
```

no need for [] when passing an array as a parameter

# Arrays Returned by Methods

- **Typical Header Syntax:**

```
        public static <type>[] <methodName>(<parameters>)
```

- **Example:**

Write a method `stutter()` that takes an integer array as an input parameter and returns a new array with two copies of each element of the original array.

```
public static int[] stutter(int[] integers) {
     int[] result = new int[2 * integers.length];
     for (int i = 0; i < integers.length; i++) {
          result[2 * i] = integers[i];
          result[2 * i + 1] = integers[i];
     }
}
```

# stutter(): Complete Example

```
public class Stuttering {
      public static int[] stutter(int[] integers) {
            int[] result = new int[2 * integers.length];
            for (int i = 0; i < integers.length; i++) {
                  result[2 * i] = integers[i];
                  result[2 * i + 1] = integers[i];
            }
      }
      public static void main(String[] args) {
            int[] nums = {51, 22, 10, 1, 15, 6};
            int[] stuttered = stutter(nums);
            for (int i = 0; i < stuttered.length; i++)
                  System.out.print(stuttered[i] + “ ”);
      }
}
```

Output? **51 51 22 22 10 10 1 1 15 15 6 6**

# Array Restrictions Encountered So Far

- Any array declared and initialized **can no longer be resized**.
- **Example:**

```
int[] arr = new int[10];
arr.length = 15;                        // ERROR
```

- Arrays, by themselves, **cannot be compared** using == or `equals()` method.
- **Example:**

```
int[] arr1 = {1, 2, 3}; int[] arr2 = {1, 2, 3};
System.out.print(arr1 == arr2);         // false
System.out.print(arr1.equals(arr2));    // false
```

- Arrays **cannot be directly printed** to the screen:
- **Example:**

```
int[] arr1 = {1, 2, 3};
System.out.print(arr1);  // [I@5acf9800] address of arr1.
```

# Exercise 3: Test For Array Equality

Design a JAVA method called `equalArrs()` that takes two integer arrays `srcArr` and `destArr` as parameters and returns `true` if these two arrays are equal and `false` otherwise. For two arrays to be equal, they have to be of the same size and each and every element of the first array needs to be equal in value to its corresponding element in the second array located at the same index.

**Solution:**

```
public static boolean equalArrs(int[] srcArr, int[] destArr) {
   if (srcArr.length != destArr.length)
      return false;
   for (int i = 0; i < srcArr.length; i++)
      if (srcArr[i] != destArr[i])
         return false;
    return true;
  }
```

# Exercise 4: Array to String

Design a JAVA method called `toString()` that takes an integer array `arr` of an arbitrary size and returns the contents of `arr` as a `String` having the form: `[<e1>, <e2>, <e3>, ...]` where `<e1>`, `<e2>`, ... constitute the individual integer elements’ values of the array.

**Solution:**

```
public static String toString(int[] arr) {
     String arrStr = “[”;
     for (int i = 0; i < arr.length – 1; i++)
          arrStr += arr[i] + “, ”;
     arrStr += arr[arr.length – 1] + “]”;
     return arrStr;
}
public static void main(String[] args) {
     int[] ints = {1, 2, 3};
     System.out.println(“integers = ” + toString(ints));
}
```

Output? **`integers = [1, 2, 3]`**

# Exercise 5: Fill Array With Some Value

Write a JAVA method called `arrFill()` that takes two input parameters, namely: *i*) an integer array, say `arr`, and *ii*) a specific value, say `val`. Regardless of the size of `arr`, `arrFill()` will then set the value of every element of that array to the given `val`. In the main() method, test `arrFill()` by declaring an array, `myArray`, of an arbitrary `size` of your choice and, then, using this method, fill it out with some `value`.

- **Solution:**

```
public static void arrFill(int[] arr, int val) {
    for (int i = 0; i < arr.length; i++) {
        arr[i] = val;
}
public static void main(String[] args) {
    int size = 5, value = 2; int[] myArray = new int[size];
    arrFill(myArray, value);
    // Testing arrFill() by printing out the content of myArray
    for (int i = 0; i < myArray.length; i++)
        System.out.print(myArray[i] + “ ”);
}
```

# Exercise 6: Search for an Element in an Array

Write a JAVA method called `binarySearch()` that takes two input parameters, namely: *i*) a sorted integer array, say `arr`, and *ii*) a specific value, say `val`. This method will search for val in arr. If found, the index of val needs to be returned. Otherwise, a value -1 will be returned.

- **Solution:**

```
public static int binarySearch(int[] arr, int val) {
    int low = 0, high = arr.length;
    int mid = arr.length / 2;
    while (low < high) {
        if (arr[mid] == val) return mid;
        else if (arr[mid] < val) low = mid + 1;
        else high = mid - 1;
        mid = (low + high) / 2;
    }
    return -1;
}
```

# **Arrays** Class

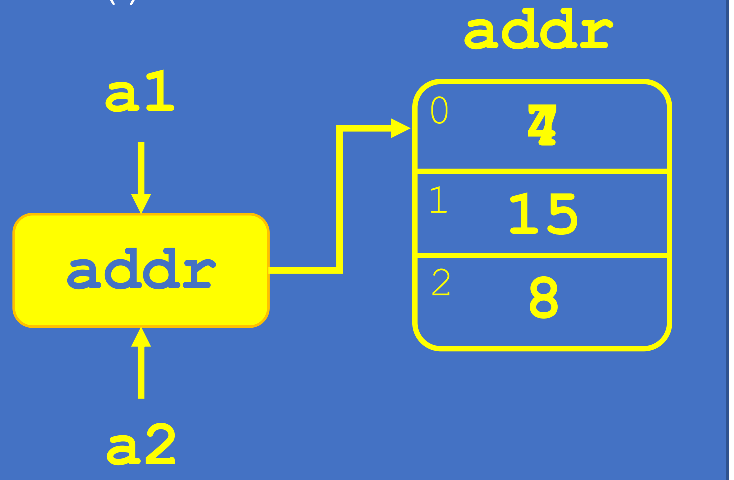
- Part of the `java.util` package.
- Has useful `static` methods for manipulating arrays. Some are:

| Method name | Description |
|---|---|
| `binarySearch(`**`sortedArray`**`, `**`value`**`)` | returns index of **`value`** in **`sortedArray`** (or **<0** if not found) |
| `copyOf(`**`array`**`, `**`n`**`)` | returns a new copy of **`array`** including the first **`n`** elements |
| `copyOfRange(`**`array`**`, `**`start`**`, `**`end`**`)` | returns a subset of **`array`** from index **`start`** to index **`end-1`** |
| `equals(`**`arr1`**`, `**`arr2`**`)` | returns `true` if **`arr1`** and **`arr2`** contain same elements in the same order. Otherwise, it returns `false` |
| `fill(`**`array`**`, `**`value`**`)` | sets every element of **`array`** to the given **`value`** |
| `sort(`**`array`**`)` | arranges the elements of **`array`** into sorted order |
| `toString(`**`array`**`)` | returns a `String` representing `array`, such as `"[<e1>, <e2>, <e3>, ...]"` |

# Arrays Class: Example 1

What will be printed to the screen following the execution of the below JAVA `main()` method:

```
public static void main(String[] args) {
        int[] a1 = {4, 15, 8};
        System.out.println(a1);
        int[] a2 = a1; // refer to same array as a1
        a2[0] = 7;
        System.out.println(Arrays.toString(a1));
}
```



**Output:**

```
[I@4b71bbc9]
[7, 15, 8]
```

printing `a1` directly will show the address of `a1`.

printing `a1` using `Arrays.toString()`.

# Arrays Class: Example 2

After importing the Arrays class, what will be printed to the screen following the execution of the below JAVA main() method:

```
public static void main(String[] args) {
      int[] a = {1, 7, 5, 6, 4, 14, 11};
      Arrays.sort(a);
      System.out.println(Arrays.toString(a));
      int index = Arrays.binarySearch(a,6);
      System.out.println(index);
      int[] b = {1, 7, 5, 6, 4, 14, 11};
      System.out.println(Arrays.equals(a,b));
      Arrays.fill(a, 2);
      System.out.println(Arrays.toString(a));
      int[] c = Arrays.copyOf(b,4);
      System.out.println(Arrays.toString(c));
}
```

**Output:**

```
[1, 4, 5, 6, 7, 11, 14]
3
false
[2, 2, 2, 2, 2, 2, 2]
[1, 7, 5, 6]
```

# Reference Semantics

# swap() Method – Is This Valid? Why?

```
public static void main(String[] args) {
          int a = 7, b = 35;
          // swap a with b?
          swap(a, b);
          System.out.println(a + " " + b);
}
public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

- swap() fails at swapping the values of a and b in main():
  1. Return type is void.
  2. Even if return type is not void, swap() cannot return two values.
  3. swap() cannot escape from itself to modify variables outside its scope.

# Value Semantics

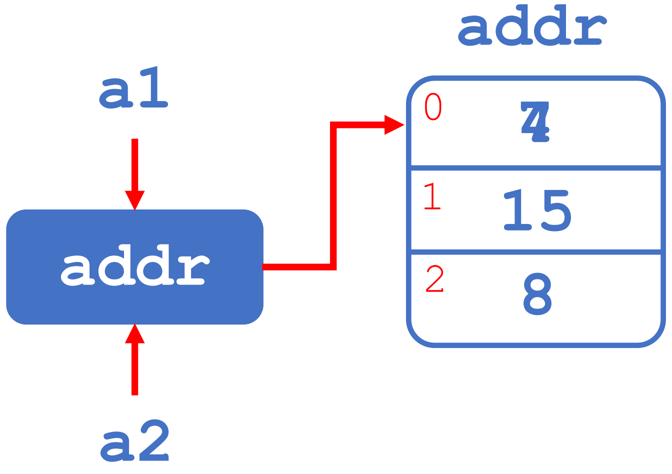
- Behavior where values are copied when:
  - **Assigned.**
  - **Passed as parameters.**
  - **Returned.**
- All primitive types in JAVA use value semantics:
  - Variables **store primitive values directly**.
  - When one variable is assigned to another, its **value is copied**.
  - Modifying the value of one variable **does not affect others**.
- **Example:**

```
int x = 5;
int y = x;      // x = 5, y = 5
y = 17;         // x = 5, y = 17
x = 8;          // x = 8, y = 17
```

# Object Reference Semantics

- Behavior where variables actually **store the address of an object in memory**.
- When one variable (*i.e.* **assigner**) is assigned to another (*i.e*. **assignee**):
  - The object is **not copied**.
  - The **reference** (*i.e.* address) in the assigner **is copied** into the assignee.
  - Both variables refer to the **same object** (*i.e.* **aliasing**)
  - Modifying the value of one variable **will** affect others.
- **Example:** recall the below example:

```
int[] a1 = {4, 15, 8};
int[] a2 = a1;
a2[0] = 7;
System.out.println(Arrays.toString(a1));
```



Output? [7, 15, 8]

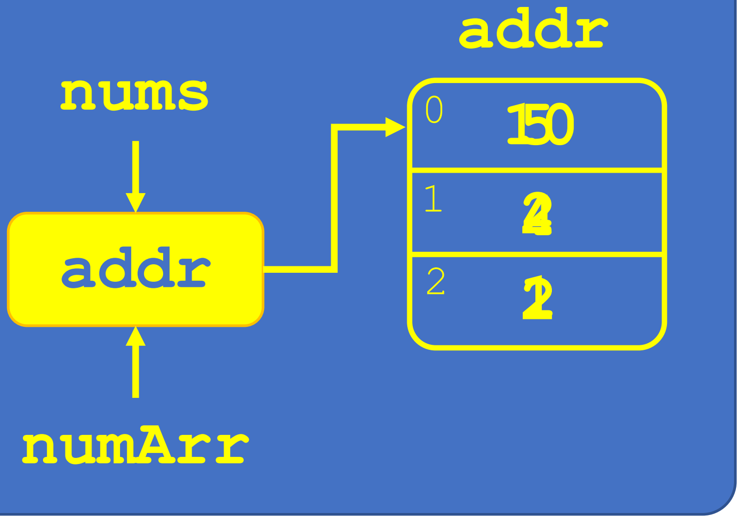
# Arrays Pass By Reference

- So far, arrays:
  - Have been passed to methods as input parameters.
  - Habe been returned by methods as end results.
- None of the developed methods so far has modified an existing array.
- Actually, arrays are passed to methods as parameters **by reference**:
  - **Two different enviroments:** caller and callee (*i.e.* method).
  - **Two independent variables:** one in caller scope, another in method scope.
  - Both variables **pointing to the same array object** (*i.e.* aliases).
  - **Changes made using the callee's (method) variable are visible to the caller.**

# Example: Arrays Pass By Reference

Consider the below `main()` method calling a method `doubleVals()` intended to double the value of each element in an array. Design and fill in the blanks so that `doubleVals()` serves this purpose.

```
public static void main(String[] args) {
      int [] nums = {5, 2, 1};
      doubleVals(nums);
      System.out.println(Arrays.toString(nums));
}
public static ____ doubleVals(________) {
      _________________________________
}
```



**Solution:**

```
public static void doubleVals(int[] numArr) {
      for(int i = 0; i < numArr.length; i++)
            numArr[i] *= 2;
}
```

# Exercise 7: Recursive Array Reverse

Develop a recursive JAVA method called `swapElements()` that accepts an array of integers together with two indices and swaps the elements at those indices. No new array declaration is allowed. `swapElements()` shall return no result to its calling point.

- **Solution:**

```
public static void swap(int[] numArr, int i, int j) {
     int temp = numArr[i];
     numArr[i] = numArr[j];
     numArr[j] = temp;
}
```

# Exercise 8: Recursive Array Reverse

Develop a recursive JAVA method called `recArrRev()` that recursively reverses the elements in an integer array. The method must work for arrays of any size. Carefully choose the method’s input parameters. No new array declarations are permitted. `revArrRev()` must not return any result to its calling point.

- ***Hint***: Think of swapping edge elements and work inwads.
- **Solution:**

```
public static void reverse(int[] numArr, int i, int j) {
    if (i < j) {
        swapElements(numArr, i, j); // see Exercise 7.
        reverse(numArr, i + 1, j - 1);
    }
}
```

# Practice Exercise: Iterative Array Reverse

- In light of Exercise 7:
  - Write an iterative version of `recArrRev()`, namely, `iterArrRev()`.
  - This iterative version needs to iteratively reverse the elements of an array.
  - It is not allowed to create any additional array.
  - `iterArrRev()` must not return any result to its calling point.

# Exercise 9: Merging Arrays

Write a JAVA program called `ArrayMerge.java` that incorporates a method called `merge()`. This method accepts two arrays of integers and returns a new array containing all elements of the first array followed by all elements of the second.

**Solution:**

```
public class ArrayMerge {
  public static void main(String[] args) {
    int[] arr1 = {12, 34, 56}; int[] arr2 = {7, 8, 9, 10};
    int[] arr3 = merge(arr1, arr2);
    System.out.println(Arrays.toString(arr3));
  }
  public static int[] merge(int[] a1, int[] a2);
    int[] result = new int[a1.length + a2.length];
    for (int k = 0; k < a1.length; k++) result[k] = a1[k];
    for (int k = 0; k < a2.length; k++) result[a1.length + k] = a2[k];
    return result;
  }
}
```

# Arrays for Tallying

# Tally

- **Meaning:** compute the total number of ... (something).
- **A Programming Example:**
  - **Task:** Given a number `n`, determine the digit that occurs most frequently in `n`.
    - If a tie exists, return the digit with the smaller value.
  - Write a method `mostFrequentDigit()` to perform the above task.
  - **Sample:**
    - The number `669260267` contains one `0`, two `2`, four `6`, one `7` and one `9`:
      - Here `mostFrequentDigit()` returns `6`.
    - The number `57135203` contains one `0`, one `1`, one `2`, two `3`, two `5`, one `7`:
      - Here the tie between `3` and `5` leads `mostFrequentDigit()` to return `3`.
  - **Silly (Bad) Solution:**
    - Declare 10 `int` counters, one for each digit.
    - Scan `n` and increment the appropriate counter corresponding to each ecountered digit in `n`.

# Tally

- **Better Solution:**
  - Use an array `digitCounters` of size `10`.
  - The element at index `i` will store the counter for digit value `i`.
  - Scan `n` and increment the appropriate element in `digitCounters`.
  - **Sample:** for the number `669260267`

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **digitCounters** | 1 | 0 | 2 | 0 | 0 | 0 | 4 | 1 | 0 | 1 |

**Q:** How to build such an array as `digitCounters`?

# Tally Solution

**n = 669260267**

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **digitCounters** | 1 | 0 | 2 | 0 | 0 | 0 | 4 | 1 | 0 | 1 |

**Q:** How to build such an array as `digitCounters`?

```
public static int mostFrequentDigit(int n) {
   int[] digitCounters = new int[10]; int digit, bestIndex = 0;
   // pluck off a digit and add to proper counter
   while (n > 0) {
      digit = n % 10; digitCounters[digit]++; n /= 10;
   }
   // find the most frequently occurring digit and break tie
   for (int i = 1; i < digitCounters.length; i++)
      if (digitCounters[i] > digitCounters[bestIndex])
           bestIndex = i;
   // return most frequent digit
   return bestIndex;
}
```

# Practise Exercise: Generalized Tally Solution

Generalize the previously developed solution for any number `n` be it an integer or a floating point number.

# Variable Length Parameter Lists

# Variable Length Parameter Lists (VLPLs)

- **Problem:**
  - Design a method that processes different amounts of data per invocation.
- **Example:**
  - Recall the `average()` method to compute the mean of an array's elements:
    - This required that the number of elements in the array be known a-priori.
  - This method may be designed in a different way:
    - To overcome the a-priori knownledge of the number of elements that it will take.
  - **Example Call 1:** `double m1 = average(42, 69, 37);`
  - **Example Call 2:** `double m2 = average(35, 43, 29, 30, 15);`
  - Here, numbers are passed directly rather than packaging them in an array:
    - Make it the method's job to package the elements in an array of appropriate size.
- **Syntax:**

```
public static <type> <name>(..., <l_type> ... <paramList>)
```

Other singly parameters and their types

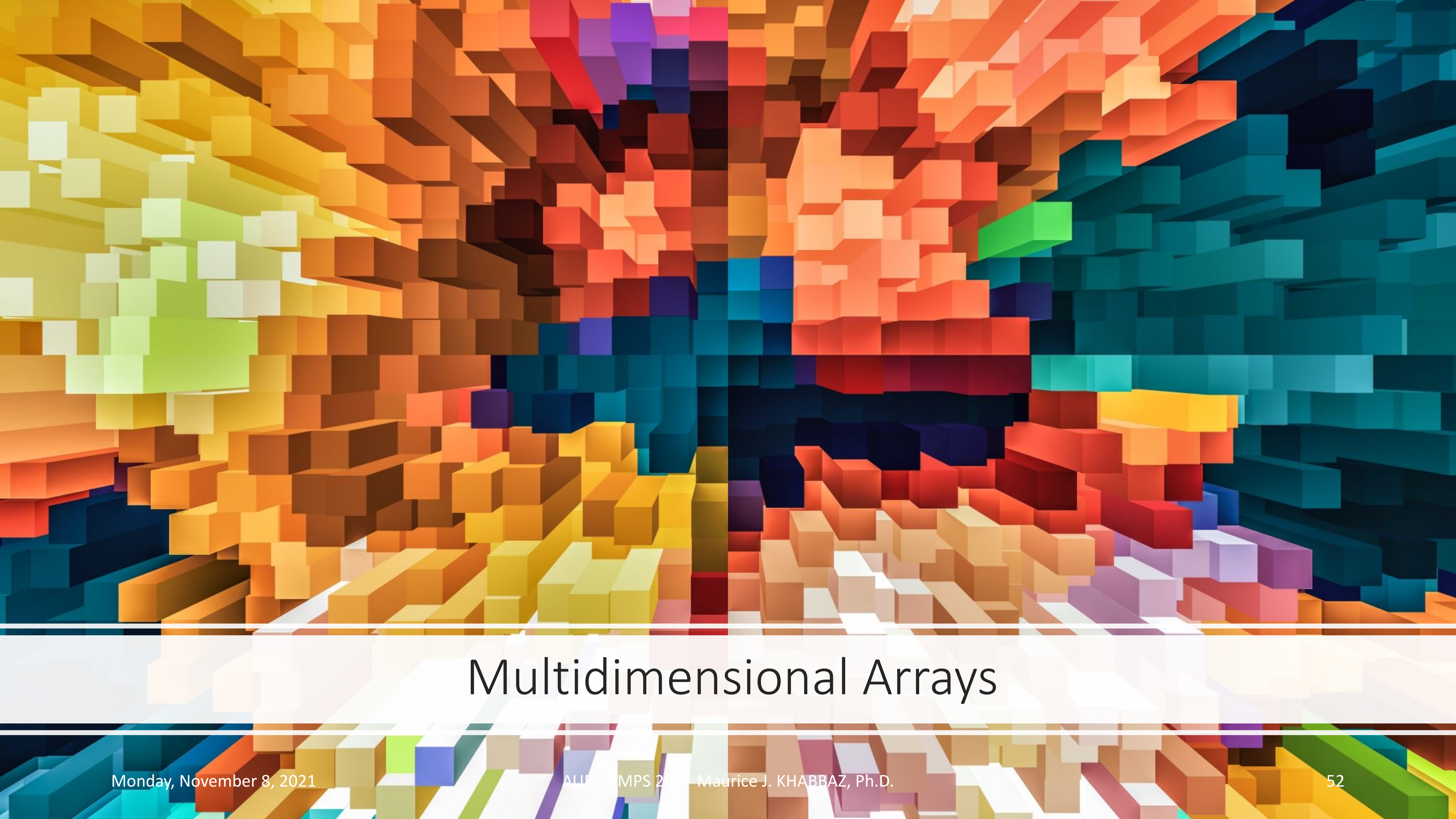
- VLPL comes last in the list of parameters.
- Method can accept only one VLPL.

# Exercise 10: Variable Length Parameter List

Implement the `average()` method in such a way that it takes a variable length parameter list of integers as an input and returns their average.

**Solution:**

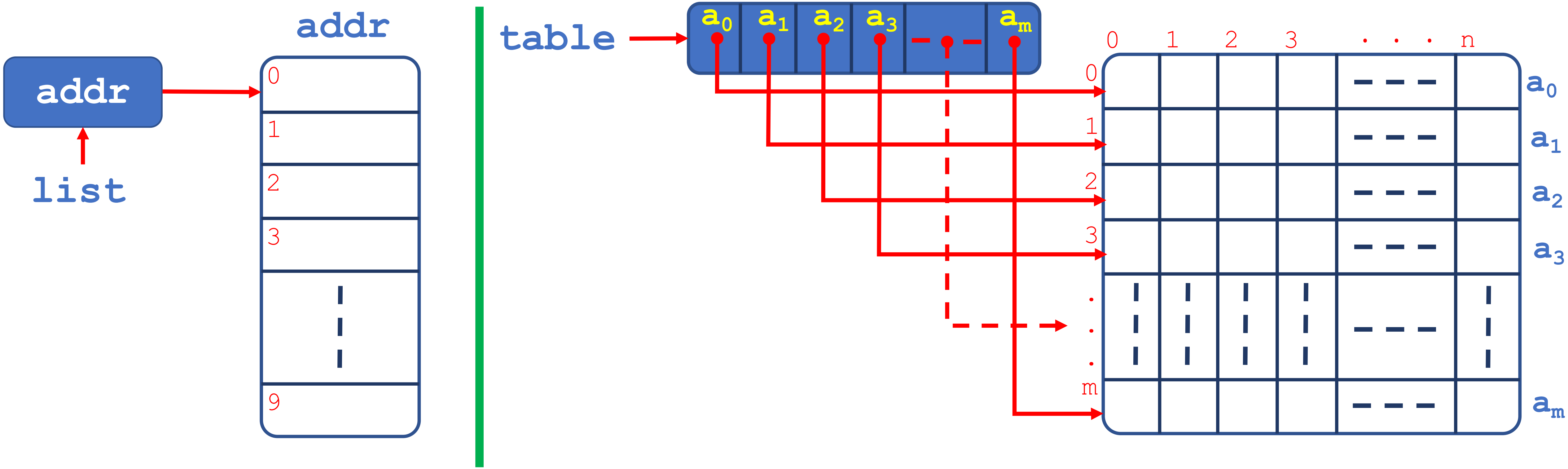
```
public static double average(int ... numbers) {
    double result = 0.0;
    if (numbers.length != 0) {
        for (int n: numbers)
            result += n;
        result /= numbers.length;
    }
    return result;
}
```



# Multidimensional Arrays

# Two-Dimensional (2D) Arrays

- Higher dimensions may be captured in JAVA through various data structures.
- **Recall:** a regular array stores a list of elements (*i.e.* one-dimensional (1D) array)
- A 2D array can be interpreted as a table of elements with rows and columns:
  - More precisely, in JAVA this is an array of arrays.



# Two-Dimensional (2D) Arrays

- **Declaration:**
  - Specify the type, name and size of each dimension (rows, columns).
  - Here, an element will be referenced by two indices.
  - An array stored at one specific row is referenced using only the row index.
- **2D Array Declaration Syntax:**

```
<a_type>[][] <a_name> = new <a_type>[<rows>][<cols];
```

- **2D Array Element Referencing Syntax:**

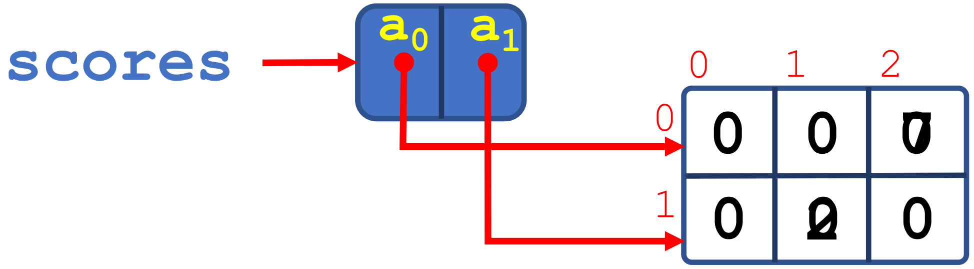
```
<a_name>[<r_index>][<c_index>]
```

- **1D Sub-Array Referencing Syntax:**

```
<a_name>[<r_index>]
```

# 2D Arrays - Example 1

```
int[][] scores = new int[2][3];
scores[0][2] = 7;
scores[1][1] = 2;
```



- **Q:** State the type and describe the reference pointed by each expression below:

| Expression | Type | Description |
|---|---|---|
| `scores` | `int[][]` | 2D array of integers, or array of integer arrays |
| `scores[0]` | `int[]` | Array of integers |
| `scores[1][2]` | `int` | An integer |

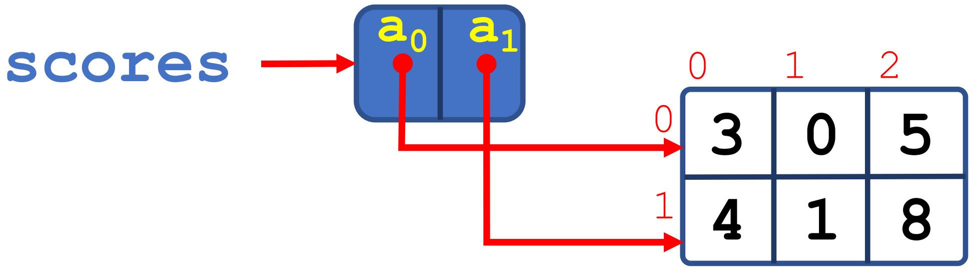
# 2D Array Initializer

- **Initializer List 2D Array Declaration/Initialization Syntax:**

```
<type>[][] <a_name> = { {<list1>}, {<list2>}, ...};
```

- **Example:**

```
int[][] scores = { {3, 0, 5}, {4, 1, 8} };
```

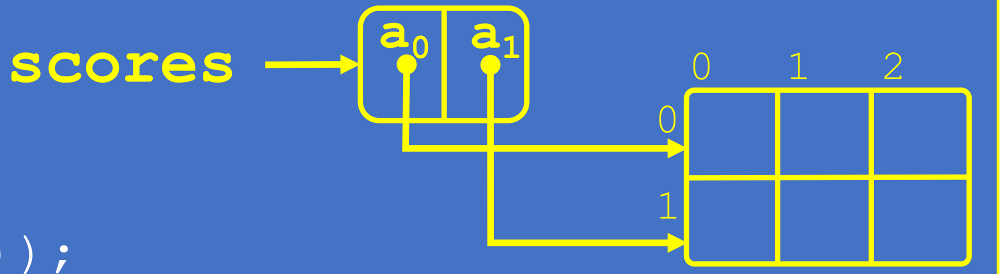


|   | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 3 | 0 | 5 |
| 1 | 4 | 1 | 8 |

# Exercise 11: 2D Arrays Tracing Outputs

Following the execution of the below JAVA code, what is the produced output to the screen?

```
public static void main(String[] args) {
        double[][] table = new double[2][3];
        System.out.println(table);
        System.out.println(table[0]);
        System.out.println(Arrays.toString(table));
        System.out.println(Arrays.toString(table[0]));
        System.out.println(Arrays.deepToString(table));
        System.out.println(table.length);
        System.out.println(table[0].length);
}
```



**OUTPUT**

```
[[D@5acf9800
[D@4617c264
[[D@4617c264, [D@36baf30c]
[0.0, 0.0, 0.0]
[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]
2
3
```

# 2D Arrays As Method Input/Output Parameters

- **Method header syntax – 2D Array as input parameter:**

```
public static <type> <name>(<a_type>[][] <a_name>)
```

- **Method header syntax – 2D Array as returned parameter:**

```
  public static <a_type>[][] <name>(<parameters>)
```

# Exercise 12: Printing a Grid of Numbers

Write a JAVA method called `printGrid()` that takes a 2D array of floating point numbers and prints them out in a grid format with each number rounded to 1 decimal digit after the decimal point and all numbers printed using six characters wide being aligned to the left.

**Solution:**

```
public static void printGrid (double[][] grid) {
     for(int i = 0; i< grid.length; i++) {
          for(int j = 0; j < grid[i].length; j++)
               System.out.printf("%-6.1f",grid[i][j]);
          System.out.println();
     }
}
```

# Exercise 13: Matrix Addtion

Write a JAVA method called `matAdd()` that takes two matrices of floating point numbers, say `M1` and `M2`. The method then creates and returns a new result matrix R, that contains the addition of the elements of `M1` and `M2`. Matrix addition is an element-by-element operation; meaning that an element of one matrix is added to the corresponding element of the second matrix. Hence, the two input matrices have to be of equal sizes.

**Solution:**

```
public static double[][] matAdd(double[][] M1, double[][] M2){
      int r1 = M1.length, c1 = M1[0].length,
         r2 = M2.length, c2 = M2[0].length;
      double[][] R = null; // in case M1 and M2 are of different sizes.
      if (r1 == r2 && c1 == c2) {
            R = new double[r1][c1];
            for (int r = 0; r < r1; r++)
                  for (int c = 0; r < c1; c++)
                        R[r][c] = M1[r][c] + M2[r][c];
      }
      return R;
}
```

# Practice Exercise: Matrix Transpose

Write a JAVA method called `matTranspose()` that take an (`m×n`) matrix `M` as a parameter and returns its transposed version. The transposed version of `M` is a matrix `MT` where the rows elements of `M` become the columns elements of `MT`.

**Example:**

$$M = \begin{bmatrix} 1 & 0 & 5 \\ 2 & 4 & 3 \\ 6 & 8 & 9 \end{bmatrix} \Rightarrow MT = \begin{bmatrix} 1 & 2 & 6 \\ 0 & 4 & 8 \\ 5 & 3 & 9 \end{bmatrix}$$

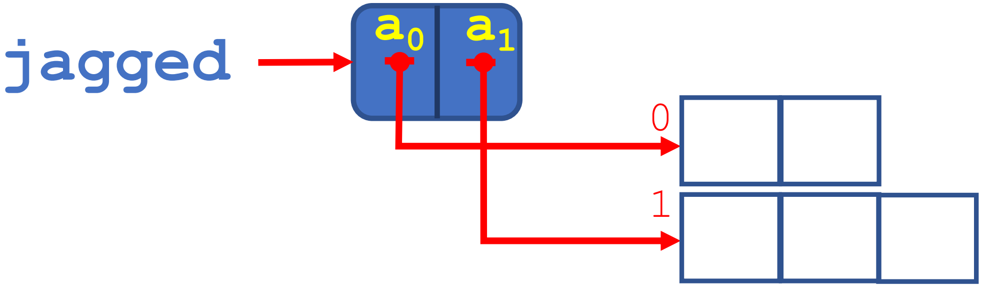
# Jagged Array

- An array of arrays such that **member arrays are of different sizes**.
- The **number of rows** of a jagged array is **fixed at declaration time**.
- The **number of columns is left undefined**:
  - Hence, at declaration time, the **jagged array elements are initialized to `null`**.
- Obviously, a jagged array's elements are **reference types**.
- **Syntax:**

```
<ja_type>[][] <ja_name> = new <ja_type>[<rows>][];
```

- **Example:**

```
int[][] jagged = new int[2][];
jagged[0] = new int[2];
jagged[1] = new int[3];
```



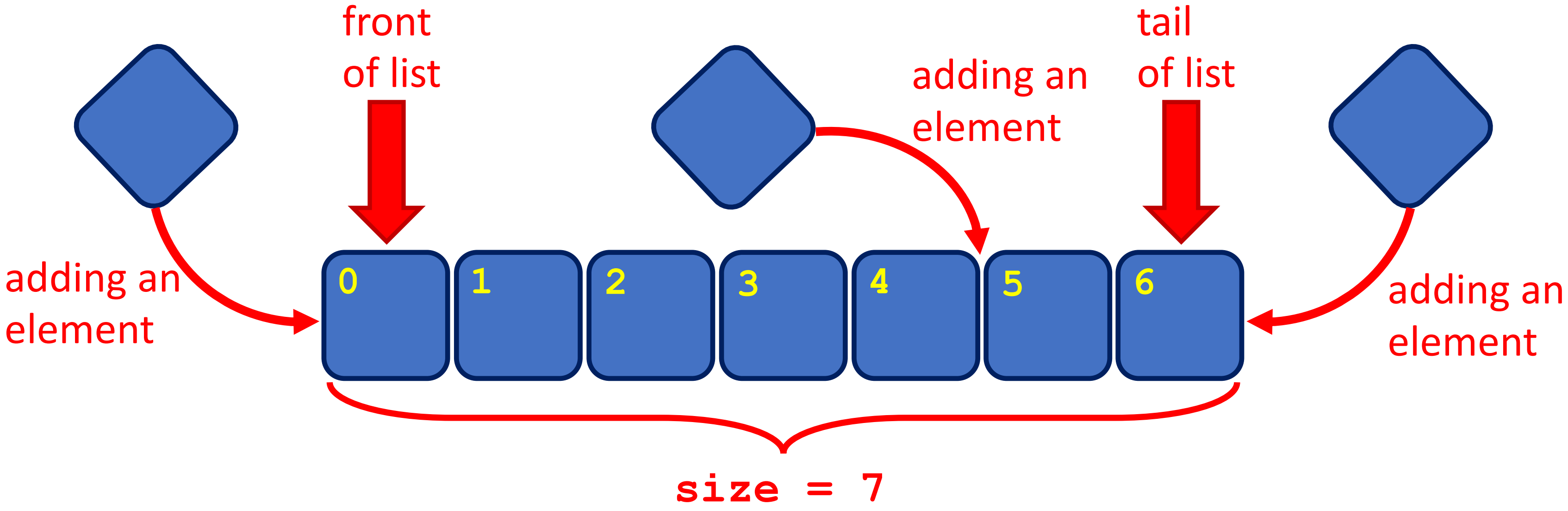
# Collections

# Introduction to Collections

- **Collection:** an object that stores data (*a.k.a.* a data structure).
- **Elements:** stored data objects.
- Some collections maintain **ordering** and allow **duplicates**.
- **Typical Operations:**
  - **Add** an element.
  - **Remove** an element.
  - **Delete** an element.
  - **Clear** the entire collection.
  - **Search** for an element.
  - Determine the **size** (*i.e.* number of elements).
- Famous collections defined in built-in classes:
  - `ArrayList`, `LinkedList`, `HashMap`, `TreeSet`, `PriorityQueue`, `...`
  - All of these collection classes are part of the `java.util` package (need to be imported).

# List

- A collection storing an **ordered sequence** of elements.
- Each element is **accessible** by a zero-based index.
- A list has a **size** (number of elements it contains).
- Elements can be **added** to the front, tail (*i.e.* rear/back) or elsewhere.
- In JAVA one way to represent a list is through an **`ArrayList`** object.



# ArrayList Class

# Idea Behind `ArrayList`

- `ArrayList` is a class that **allows the collection of objects** just as arrays do.
- As opposed to arrays, `ArrayList` has **three notable conveniences**:
  1. Initially an `ArrayList` is **empty**.
  2. An `ArrayList` can **grow and shrink as needed**.
     - Think of it as an **automatically resizable array** object.
     - Internally the list is implemented using an array and a size field.
  3. `ArrayList` class **defines methods for many common tasks**:
     - **Example:** adding or removing items
     - By default any new items are added (*i.e.* appended) to the end of the list.
- Also an `ArrayList` **keeps track of items** added to it in terms of:
  - **Order**.
  - **Indices**.
  - **Total Size**.

# `ArrayList`: Generic Syntax

```
ArrayList<type> <name> = new ArrayList<type>();
```

- When constructing an `ArrayList`, its items' `type` must be specified.
- The `type` must be included in between `<` and `>` (these are part of the syntax):
  - This is called the **type parameter** or **generic class**.
- Upon construction, the `ArrayList` size is `0`:
  - The size increases each time an item is added to the `ArrayList`.
  - When item is inserted in the middle, other items move to make room for it.
  - When item is removed, list collapses (items move down by one position) to close the gap.
  - Opposite to `length` for arrays, the `size()` method → the # of items in an `ArrayList`.
  - Items in an ArrayList are referenced using indices from `0` to `size()-1`.

# Valid `ArrayList` Types

- The `type` specified for an `ArrayList` **cannot be a primitive data type**:
  - It has to be an **object type**.
- **Problem:** Primitive data types cannot be directly inserted into an `ArrayList`.
- **Solution:** Use Wrapper Classes!
  - Recall, **wrappers** are objects having the sole purpose of holding a single primitive value.
- **Example:**

```
 // invalid -- int cannot be a type parameter
     ArrayList<int> list = new ArrayList<int>();
// valid -- instantiates a list of ints
     ArrayList<Integer> list = new ArrayList<Integer>();
```

# ArrayList Methods

| Method | Description |
|---|---|
| add(**value**) | appends value at end of list |
| add(**index, value**) | inserts value just before the index, shifting subsequent values to the right |
| clear() | removes all elements of the list |
| indexOf(**value**) | returns first index where given value is found in list (-1 if not found) |
| get(**index**) | returns the value at given index |
| remove(**index**) | removes/returns value at given index, shifting subsequent values to the left |
| set(**index, value**) | replaces value at given index with given value |
| size() | returns the number of elements in list |
| toString() | returns a string representation of the list such as "[3, 42, -7, 15]" |
| contains(**value**) | returns true if given value is found in the list. Otherwise, returns false. |
| lastIndexOf(**value**) | returns last index where value is found in list (-1 if not found) |
| isEmpty() | Returns true if the list contains no elements at all. Otherwise, returns false. |

# ArrayList Example 1: add()/set() Methods

```
ArrayList<String> names = new ArrayList<String>();
System.out.println("Names = " + names);
names.add("Marty Stepp"); System.out.println(“Names = " + names);
names.add("Stuart Reges"); System.out.println("Names = " + names);
names.add(0, "Hello"); System.out.println("Names = " + names);
names.add(2, "and"); System.out.println("Names = " + names);
names.set(2, "or"); System.out.println("Names = " + names);
```

**OUTPUT**

```
Names = []
Names = [Marty Stepp]
Names = [Marty Stepp, Stuart Reges]
Names = [Hello, Marty Stepp, Stuart Reges]
Names = [Hello, Marty Stepp, and, Stuart Reges]
Names = [Hello, Marty Stepp, or, Stuart Reges]
```

# ArrayList Example 2: get()/Remove() Methods

```
ArrayList<String> names = new ArrayList<String>();
names.add("Marty Stepp"); names.add("Stuart Reges");
names.add(0, “Hello"); names.add(2, “and");
names.set(2, “or");
System.out.println(“Names = ” + names);
System.out.println(names.get(2));
names.remove(2); System.out.println(names.get(2));
System.out.println(“Names = ” + names);
```

**OUTPUT**

```
Names = [Hello, Marty Stepp, or, Stuart Reges]
Or
Stuart Reges
Names = [Hello, Marty Stepp, Stuart Reges]
```

# ArrayList V.S. Regular Array

- **Construction:**

  **Regular Array**: `String[] names = new String[5];`

  **ArrayList**: `ArrayList<String> list = new ArrayList<String>();`

- **Storing a value:**

  **Regular Array**: `names[0] = "Jessica";`

  **ArrayList**: `list.add("Jessica");`

- **Retrieving a value:**

  **Regular Array**: `String s = names[0];`

  **ArrayList**: `String s = list.get(0);`

- **Checking whether a certain value (*e.g.* "Benson”) is found:**

  **Regular Array**:

  ```
  for (int i = 0; i < names.length; i++) {
      if (names[i].equals("Benson")) { ... }
  }
  ```

  **ArrayList**: `if (list.contains("Benson")) { ... }`

# ArrayList As Method Parameter

- **Method header syntax – ArrayList as input parameter:**

```
public static <type> <name>(ArrayList<al_type> <al_name>)
```

- **Method header syntax – ArrayList as returned parameter:**

```
public static ArrayList<type> <name>(<parameters>)
```

- **Example:**

```
// Search for a string target and replace it with replacement
public static void replace (ArrayList<String> list,
                            String target, String replacement){
     int index = list.indexOf(target);
     if (index>=0)
          list.set(index, replacement)
}
```

Java
TM