Fundamentals of Programming 2

Lecture 5

Aurelia Power, TU Dublin - Blanchardstown Campus, 2019

* Notes based on the Java Oracle Tutorials (2019), Deitel & Deitel (2015), and Horstman (2013)

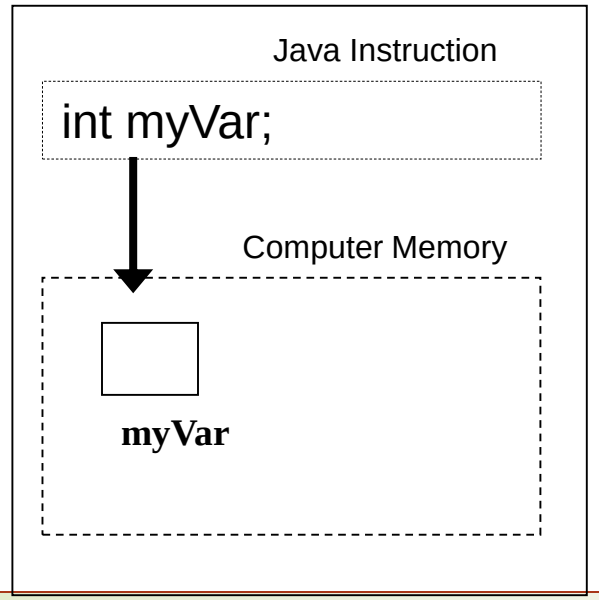
Today – Arrays

- What are arrays
- How to declare arrays
- How to instantiate arrays
- How to initialise arrays
- How to access array elements
- How to work with arrays
- How to pass arrays to methods
- How to use multi-dimensional arrays

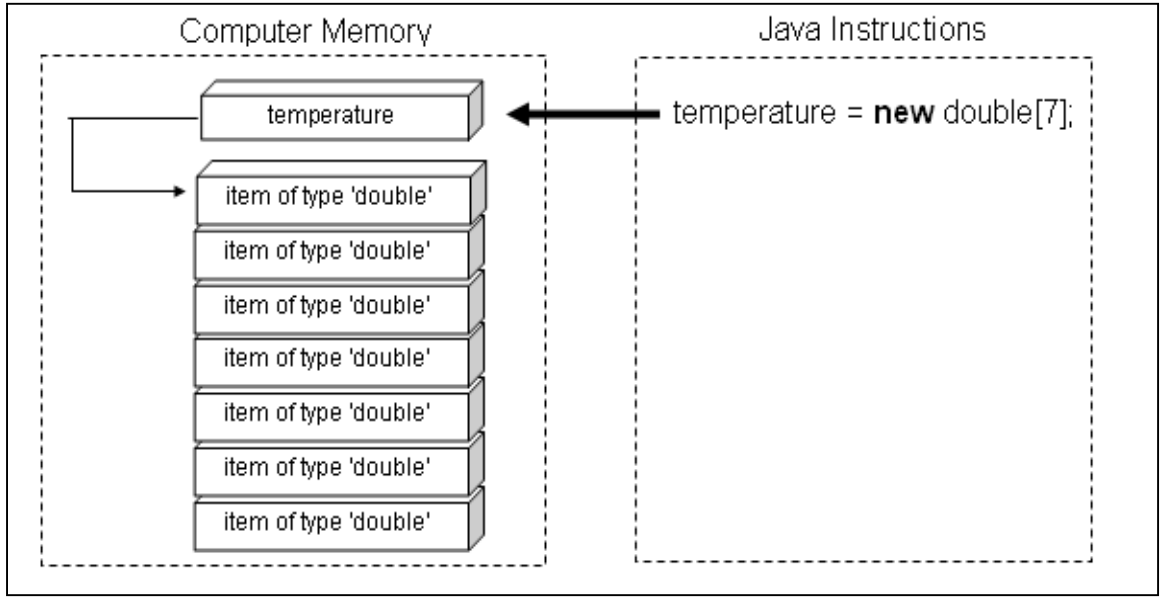
A variable vs. An Array

- A variable stores **one** value/item: int myVar;
- An array is a block of variables all of the same type that are stored as **a collection/series of values** in memory: double[] temperature;

stores a single item — **Variables**



Arrays — stores a collection/series of items



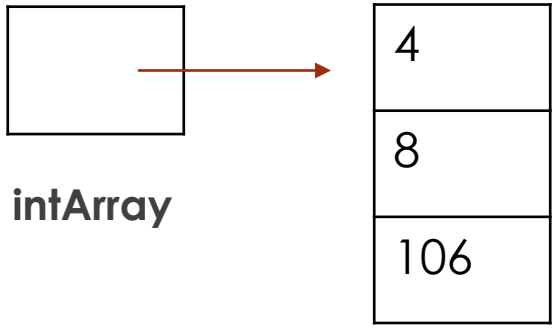
What are arrays?

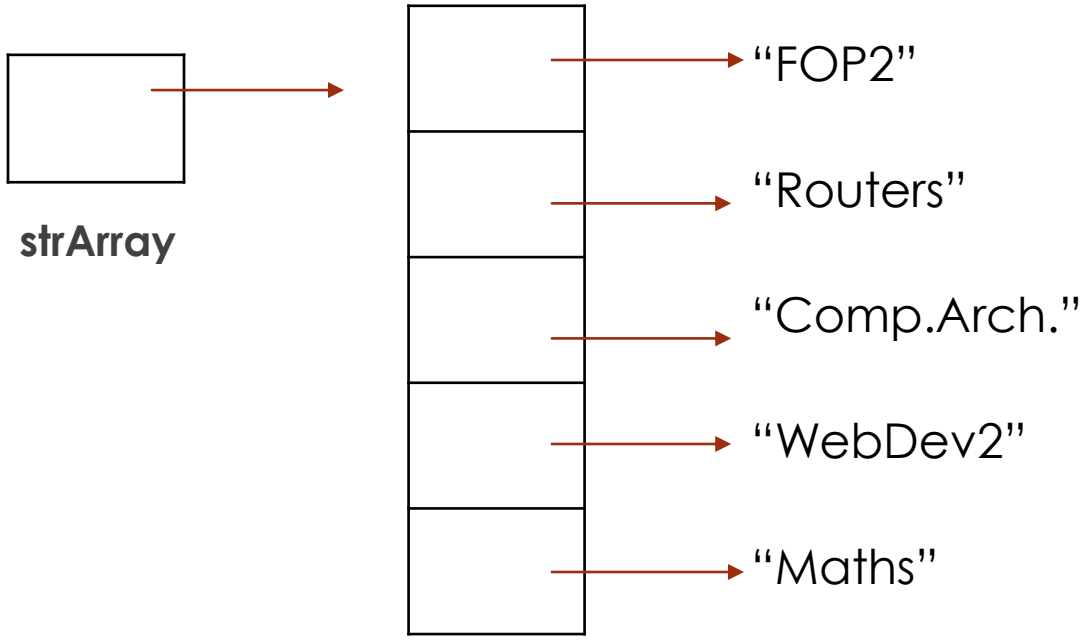
- The fundamental mechanism for **collecting multiple values**;
- Represent a data structure that stores a collection of **same type of values;**
- An array is of a **reference data type**, not primitive data type;
- But an array can hold/store both:
  - A collection of primitive data types: for example, an array of int values;
  - A collection of object references: for example, an array of String values;
- The members or elements of the arrays are stored **contiguously in memory,** indexed starting from 0.

DEFINITION: Arrays are data structures that store a collection of same type values (primitive or object references) contiguously in memory.

Arrays Length

- All arrays have a public instance variable called **length** which stores the number of elements/size of the array.





Primitive data array:

```
int[] intArray = new int[] {4, 8, 107};

System.out.println(intArray.length);
//prints 3
```

Once declared, the legth of an array cannot be changed!!!!

Array of object references:

```
String[] objArray = {"FOP2", "Routers", "Comp.Arch.",
"WebDev2", "Maths"};

System.out.println(objArray.length);
//prints 5
```

Your Turn…

- Decide what type and size of array (if any) to use to store each of the following data sets…

  a. Money spend on lunch each day for a week
  b. your name, DOB, student number and course
  c. exam letter grades e.g. A, B, C… for a class of 22 students

Syntax Overview of Creating Arrays

There are 2 main ways in which you can create an array

1. declare an array and initialise its values to default values, then access each element to change/re-assign a different value later: see the example of values
2. declare an array and assign it straight away initial custom values: see the example of moreValues

Syntax

To construct an array:	new *typeName*[*length*]
To access an element:	*arrayReference*[*index*]

Name of array variable

Element type

Length

Type of array variable

```
double[] values = new double[10];

double[] moreValues = { 32, 54, 67.5, 29, 35 };
```

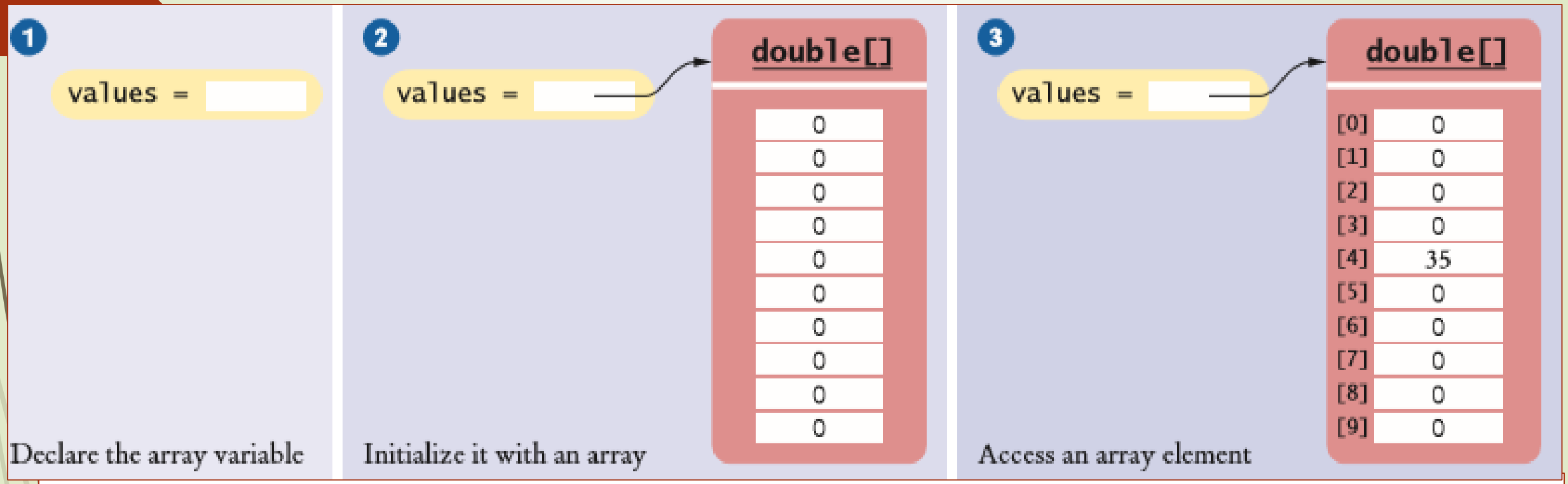
List of initial values

Use brackets to access an element.

```
values[i] = 0;
```

The index must be ≥ 0 and < the length of the array.

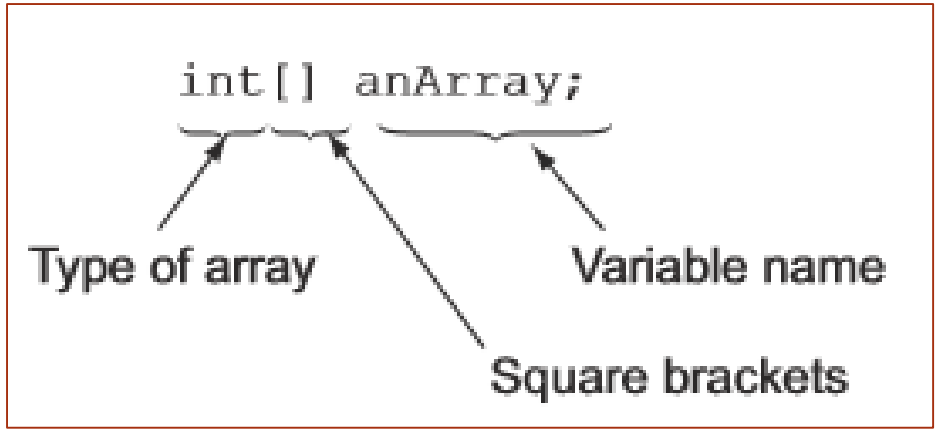
Let's see how method 1 works!!



1. **Declare an array of type double:** double [] values;
2. **Allocate memory and initialise it to default values:** values = new double[10]; <u>in this step, the data type that you declared your array to hold, must match the one that you are trying to create…</u>
3. **Access an element and assign it a particular value:** values[4] = 35; <u>in this step, the value that you are trying to assign to the array element must match the one that you declared your array to hold.</u>

Declaring arrays in more detail…

- An **array declaration** includes:
  - The array type;
  - A pair (or more if multi-dimensional) of square brackets;
  - The variable name;



- NOTE: the square brackets can be placed after or before the name, but always after the data type.
- Examples of valid declarations: **int intArray[]; String[] strArray; int[][] twoDArray; double doubleValues[]; boolean boolArray[][]; etc.**

Memory Allocation and Initialisation to Default Values

- Array allocation will **allocate memory for the elements** of the array.
- When allocating memory you **must specify its dimensions/length/the number of elements** (unless you provide straight away a list of actual values, in which case the compiler infers the size of the array).
- **NOTE:** again, once memory is allocated, the size of the array or the number of elements it contains cannot be changed.
- Examples:

```
intArray = new int[2];
strArray = new String[4];
twoDArray = new int[2][3];
```

- **NOTE:** because an array is an object we use the **new** keyword to create it and allocate memory for it (we will learn about it soon…)

Memory Allocation and Initialisation to Default Values

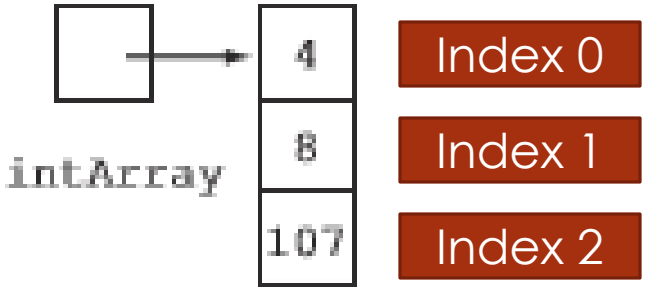
- So the syntax for allocation is: **new**, followed by the **data type**, and then the **size** specified inside the square brackets;
- the data type must match the one used in the declaration;
- The size or the number of elements must always be an **integer value**;
- Once allocated, all the array elements will be **initialised to their default values**:
  - 0 for integer types
  - 0.0 for decimal types
  - false for Boolean
  - /u0000 for char
  - null for any object reference (whether String, Random, Scanner, etc.)
- **NOTE:** for multidimensional arrays you only need to specify the size only for the first pair of square braces.
- **EXERCISE: declare an array of characters of size 7 and allocate memory for its elements. What will be the values of its elements???**

Your turn… Identify which declarations and/or allocations are valid

- `[]boolean boolArr;` // not valid: cannot place the brackets before the datatype
- `Double[] dArr;` // valid
- `float fArr[];` // valid
- `int[] array = new int[];` // not valid: we didn't specify the length/size of the array
- `int [] array = new int[2.4];` // not valid: the size/number of elements cannot be a double
- `String[] strArray = new String[2*7];` // valid
- `double[] dArray = new double[Math.max(2, 3)];` // valid
- `int[] dArray = new double[5-4];` // not valid: the datatypes do not match
- `int[] multiArr[];` // valid
- `int [][] multiArr = new int[][3];` // not valid: we must specify the number of elements in the first brackets
- `int [][] multiArr = new int[][];` // not valid: we must specify the number of elements in the first brackets
- `int [][] multiArr = new int[5][];` // valid

Element Access and Indexing

- to access an array element we use indexing like in Python, which starts at **0** and ends with **array.length -1** ;
- **So… <u>an array index must be between 0 and less than the size of the array</u>**
- **intArray = {4, 8, 107};**



- The **size** of the array is 3 because it contains 3 elements;
- The **index range (integer range!!!)** is from 0 to 2 inclusive;
- The element at index 0 is 4;
- The element at index 2 is 107

- We can change the value of element at index 1, from **8** to **7**: **intArray[1] = 7;**
- if we try to access an element that is outside the index range we get a runtime exception called **ArrayIndexOutOfBounds:**

System.out.print(intArray[3]); //it will compile, but when you run it will end with above exception

System.out.print(intArray[-10]); //it will compile, but when you run it will end with an exception

System.out.print(intArray[1.5]); //won't compile… why???

Custom Initialisation – 2 ways:

1. **Using a loop**: for example, we can initialise an array of ints with random numbers from 1 to 100 inclusive, or using user input:

```
int array[] = new int[10];
for(int index = 0; index < array.length; index++){
        array[index] = (int)(Math.random() * 100) + 1;
}
//OR by using user input:
for(int index = 0; index < array.length; index++){
        array[index] = sc.nextInt(); //assume sc is declared and initialised
}
```

2. **Initialising one element at a time:** this option may be time-consuming for entire large arrays, so it is suitable for changing selected elements.

```
array[0] = 3;
array[7] = 4;
array[9] = 77;
```

Let's try some exercises:

//given the following array declaration and memory allocation

```
int[] anArray = new int[5];
```

1. What is its **size**? // 5
2. What is the value of element **anArray[3]**? // 0
3. What about **anArray[5]**? // there is no element at index 5
4. Next, custom initialise the array to the first 5 prime numbers, accessing one element at a time. (NOTE: some may argue that 1 is not prime, but we are not concerned with this!!!)

```
anArray [0] = 1; anArray [1] = 2; anArray [2] = 3; anArray [3] = 5; anArray [4] = 7;
```

5. Now, execute the following loop:

```
for (int i = 0; i < 2; i++) {
    anArray[4-i] = anArray[i];
}
```

- Does it compile??? // yes
- If so, what's the value of **anArray** elements?

```
anArray [0] = 1; anArray [1] = 2; anArray [2] = 3; anArray [3] = 2; anArray [4] = 1;
```

Combining declaration, allocation and custom initialisation – the 2nd main way of creating arrays

- We can combine the 3 steps of declaration, allocation and initialisation in a single step in 2 ways:

1. **Without the new keyword:**

```
int intArr[] = {0, 1};
String[] strArr = {"Anna", "John", "Hellen"};
```

2. **With the new keyword:**

```
int intArr[] = new int[] {0, 1};
String[] strArr = new String[] {"Anna", "John", "Hellen"};
```

NOTE: when combining the 3 steps we don't specify the size, otherwise your code won't compile; the reason is that the compiler infers the length/size from the number of elements within the curly braces.

```
int intArr[] = new int[2] {0, 1}; //won't compile
String[] strArr = new String[3] {"Anna", "John", "Hellen"}; //won't compile
```

Array declaration, allocation and initialisation recap…

`int[] numbers = new int[10];`	An array of ten integers. All elements are initialized with zero.
`final int LENGTH = 10;` `int[] numbers = new int[LENGTH];`	It is a good idea to use a named constant instead of a "magic number".
`int length = in.nextInt();` `double[] data = new double[length];`	The length need not be a constant.
`int[] squares = { 0, 1, 4, 9, 16 };`	An array of five integers, with initial values.
`String[] friends = { "Emily", "Bob", "Cindy" };`	An array of three strings.
⊘ `double[] data = new int[10];`	**Error:** You cannot initialize a `double[]` variable with an array of type `int[]`.

2 other common errors:

- Bounds error when trying to access a non-existent element:

int[] arr = {1, 3, 5}; System.out.print(arr[3]); // runtime exception ArrayIndexOutOfBounds

- Un-initialised array: if you try to access/modify the elements of an un-initialised array

double[] arr2; arr2[0] = 4.3; // this won't compile

A full program…

```java
public class ArrayDemo1 {

    /**
     * @param args
     */
    public static void main(String[] args) {
        //declare, allocate and initialise to default values
        double[] doubleArray = new double[5];
        //create the scanner object for user input
        Scanner sc = new Scanner(System.in);
        //prompt the user to fill the array
        System.out.println("fill the array with some values: ");
        //fill the array using a for loop
        for(int i = 0; i < doubleArray.length; i++){
            doubleArray[i] = sc.nextDouble();
        }
        //display the array using a for loop
        for(int i = 0; i < doubleArray.length; i++){
            System.out.print(doubleArray[i] + "\t");
        }
        //close the scanner
        sc.close();
    }

}
```

Sample output:

```
fill the array with some values:
3
4.5
7.7
1
2
3.0     4.5     7.7     1.0     2.0
```

The enhanced for loop … the 4th type of loop in Java

Syntax

```
for (typeName variable : collection)
{
   statements
}
```

This variable is set in each loop iteration.
It is only defined inside the loop.

An array

```
for (double element : values)
{
   sum = sum + element;
}
```

These statements are executed for each element.

The variable contains an element, not an index.

- **Exercise:** write an enhanced for loop that prints all the elements of the following array:

  String[] names = {"Sean", "Jenny", "Graham", "Jason", "Lucinda"};

Other Common array algorithms…find maximum (and minimum… as an exercise for you) value in an array… exercise: implement it as a method

```
//assume the array values is declared and initialised
double largest = values[0];
for (int i = 1; i < values.length; i++) {
    if (values[i] > largest)   {
        largest = values[i];
    }
}
System.out.println(largest);
```

Other Common array algorithms…find total and average values in an array… exercise: implement them as methods

```
//assume the array values is declared and initialised
double total = 0;
for (double element : values) {
        total = total + element;
}
System.out.println(total);
```

Array References

// look at the following code fragment

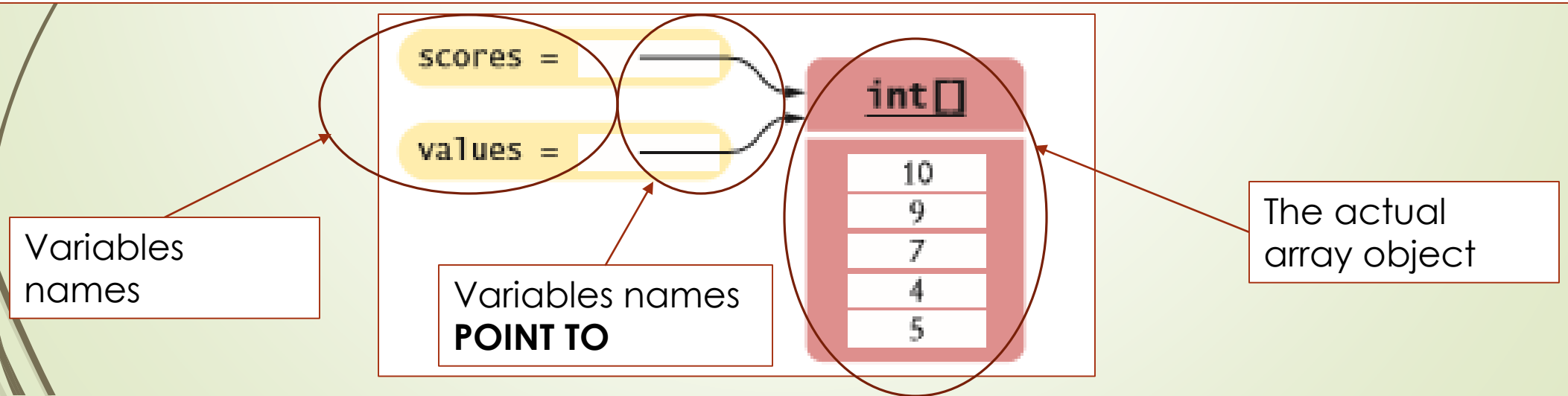
int[] scores = { 10, 9, 7, 4, 5 };

int[] values = scores;

1. What are the elements of the **values** array??
2. Are the 2 arrays **different objects** ???

Question 1 answer: 10, 9, 7, 4, 5 – same as scores array

Question 2 answer: NO!! – they refer to the same object array; in other words, even though they are 2 different variables, they point to the same object



Array References

- The variables **scores** and **values** do not store any numbers!!!
- They only hold a reference to the array!!
- The reference denotes the location of the array in memory.
- The array is stored elsewhere in memory.
- In our case, both **scores** and **values** will hold the same reference …
- So if you try to modify values using either array identifier, this will also be reflected in the other variable… because array are data structures that can be modified… (unlike strings!!!)

```
//change the value of scores[3]
scores[3] = 10;
System.out.println(scores[3]);  ///prints 10
//What about values[3]??
System.out.println(values[3]);  ///also prints 10
```

Using Arrays with methods

- Arrays can also be passed as arguments to methods:

```
//for example, a method that computes the sum of an array of double values
public static double sum(double[] values) {
double total = 0;
    for (double element : values)   {
        total = total + element;
    }
    return total;
}
```

- **Question…** are the elements of the array modified???

No… they are only accessed to find out their values and add them to the sum.

```
//in the main method… we call the method and pass an array as argument
double[] vals = {7, 3. 5, 3.5};       System.out.println(sum(vals));
```

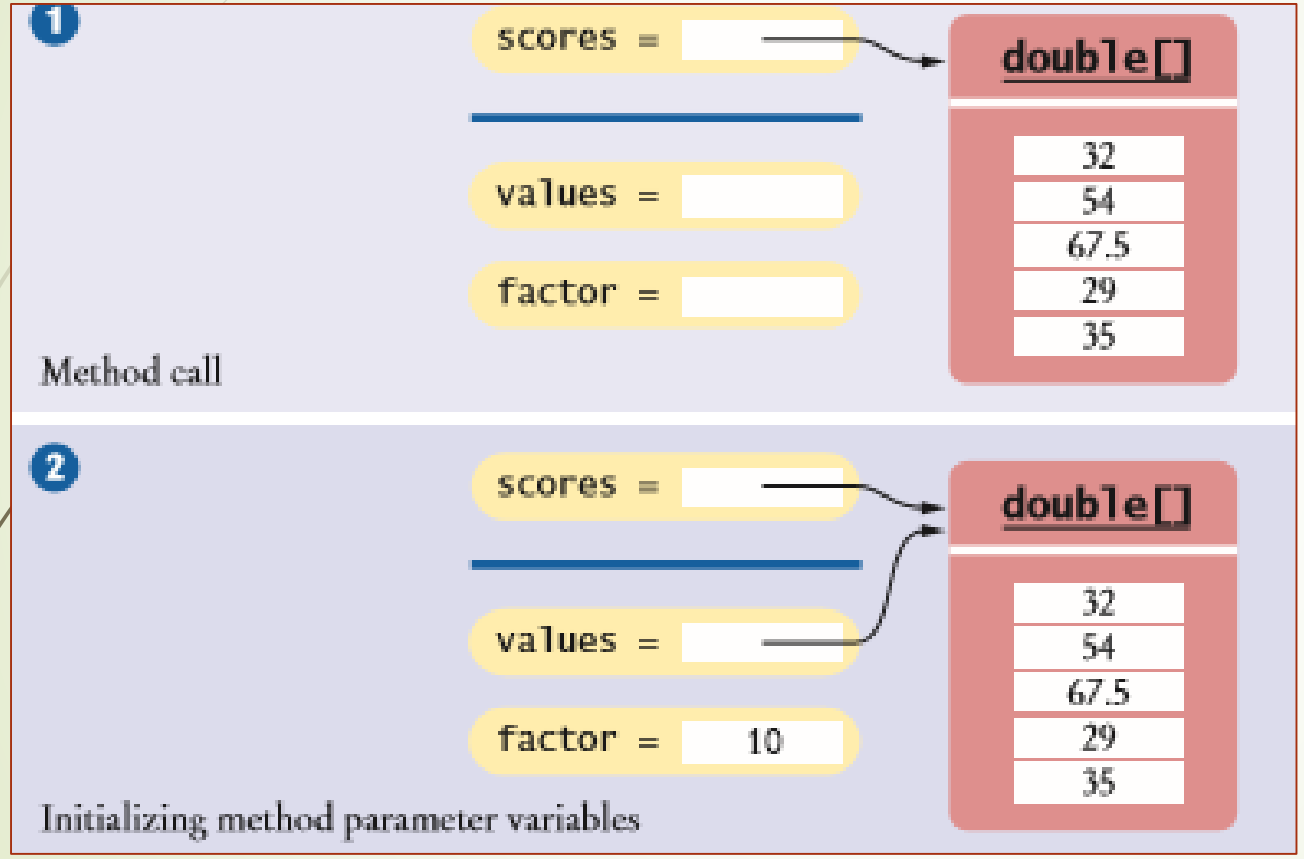
Using Arrays with methods

- A method can also **modify** the elements of an array:

```
public static void multiply(double[] values, double factor) {
    for (int i = 0; i < values.length; i++)   {
        values[i] = values[i] * factor;
    }
}
//in the main method we call the method multiply
double[] scores= {32, 54, 67.5, 29, 35};
multiply(scores, 10);
for( double score : scores){
    System.out.print(score + "\t");
}
```

OUTPUT: 320 540 675 290 350

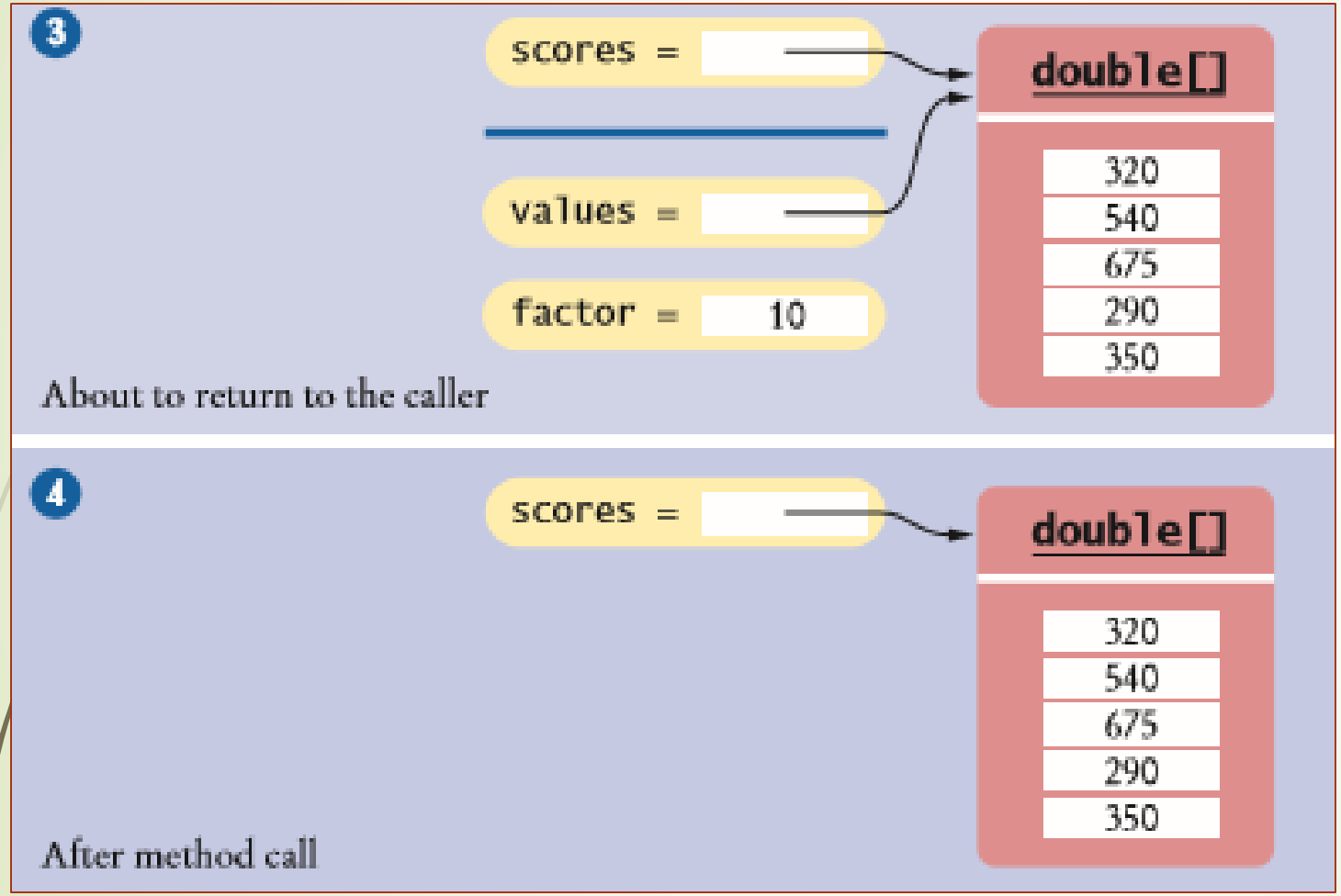
Let's trace the method call…



1. The parameter **values** and **factor** are created.

2. The parameter variables are initialised with the values of the actual arguments: **values** = **scores**, and **factor** = **10**.

Let's trace the method call…



3. The method **multiplies all the elements by 10**.

4. The method returns control to the main method and **parameter variables are destroyed** … but **scores** still refers to the array which the method has modified.

Using Arrays with methods

- A method can also **return** an array:

```
public static int[] squares(int n) {
    int[] result = new int[n];
    for (int i = 0; i < n; i++)   {
        result[i] = i * i;
    }
    return result;
}
//in the main method we call the method squares
int[] squaredValues = squares(4);
// then print its elements
for( int val : squaredValues){
    System.out.print(val + "\t");
}
```

OUTPUT: 0 1 4 9

Let's practice some more…

- Do they following methods compile, and, if so, what's their purpose?

```
public static int[] someMethod(int n) {
  int[] result = new int[n];
  for (int i = 0; i < result.length; i++)   {
        result[i] = new java.util.Random().nextInt(10);
  }
  return result;
}
```

To return an array of integer values randomly generated; the array will contain n elements, and their values will range from 0 to 9.

```
public static int[] otherMethod(int[] values) {
  int[] result = new int[values.length];
  for (int i = 0; i < values.length; i++)   {
        result[i] = values[values.length - 1 - i];
  }
  return result;
}
```

To return an array of integer values that are contained in the parameter array, but starting from the end: for instance, if values = {3, 4, 5}, the result will be {5, 4, 3}

Primitives vs. array references in methods

- There is a difference in the way arrays are treated compared to primitive data types when passing them to a method.
- In the case of **primitive data types**, it is a copy of a particular value that is passed as argument to a method…so the variable whose value was copied does not change:

```
public static void modifyVal(int a ){
    a++; System.out.println(a);
}
//in the main method….
int n = 7;
modifyVal(n); // prints 8
System.out.println(n);
/* prints 7 - because only a copy of n was passed to the method modifyVal, and the value of n remains unchanged */
```

Primitives vs. array references in methods

- In the case of **arrays** (and any other mutable object), what is passed is the reference to the array object:

```
public static void modifyArray(int[] arr ){
    for(int i = 0; i < arr.length; i++) { arr[i] = i*2; }
}
//in the main method….
int[] arr = {1, 2, 3};
modifyArray(arr);
for(int element: arr)
    System.out.print(element + " ");
//it prints: 0 2 4
```

Two-dimensional arrays

Number of rows

Name

Element type

Numberof columns

```
double[][] tableEntries = new double[7][3];
```

All values are initialized with 0.

Name

List of initial values

```
int[][] data = {
                { 16, 3, 2, 13 },
                { 5, 10, 11, 8 },
                { 9, 6, 7, 12 },
                { 4, 15, 14, 1 },
              };
```

Two-dimensional arrays

- Typically used to store tabular data, but not necessarily;
- You create a 2D array by specifying the number of rows and columns;
- A 2D array is an array of 1D arrays, all the same data type.
- To access a particular individual element you need to specify the name of the array and 2 indices: the row number first and the column number secondly:

```
int n  = data[0][1];  //we look in row 0, column 1 which is 3
```

- So the first pair of square brackets indicates the row number, while the second indicates the column number.

indices	Column 0	Column 1	Column 2	Column 3
Row 0	16	3	2	13
Row 1	5	10	11	8
Row 2	9	6	7	12
Row 3	4	15	14	1

Let's identify some more elements…

```
int data[][] = {
                {16, 3, 2, 13},
                {5, 10, 11, 8},
                {9, 6, 7, 12},
                {4, 15, 14, 1}
            };
```

1. What's the value of the following elements: **data[1][1]**, **data[4][2]**, **data[2][3]**, **data[3][0]**
2. What is the value of **data[2]?**
3. What is the value of **data[4]?**

How to access individual elements of a 2D array

- We use **2 nested loops (can use while, for, enhanced for)**:

1. The **outer loop** will access the rows or the constituent arrays (because… a 2 D array is an array of 1D arrays);
2. The **inner loop** will access each element of the constituent 1 D array, corresponding to values in each column.

```
for (int i = 0; i < data.length; i++) {   //the outer loop
    // Retrieve the ith row   in the inner loop
    for (int j = 0; j < data[i].length; j++)   {      //the inner loop
        // Retrieve the jth column in the ith row
        System.out.printf("%5d", data[i][j]);  // element in row I, column j
    }
    System.out.println(); // Start a new line at the end of the row
}
```

Computing row totals – Exercise… implement it as a method

```
for (int i = 0; i < data.length; i++) {   //the outer loop
    int rowTotal = 0;
    for (int j = 0; j < data[i].length; j++)   {      //the inner loop
        rowTotal = rowTotal + data[i][j];
    }
    System.out.println("the sum of all values in row " + i + " is: " + rowTotal);
}
```

- The outer loop will access each row at a time;
- The inner loop will access each individual value in each column in the given row, updating the value of total with each iteration
- What's the value of **rowTotal** for when i = 1??

When i = 1, it is the second iteration of the outer loop, so we look at the second component array which is **{5, 10, 11, 8}**; then the inner loop will add 5,10,11, and 8 into rowTotal; so when the inner loop is finished the value of rowTotal that is printed will be **34**

A full program…

```
public static void main(String[] args) {
    final int NUM_COLLEGES = 4;//the number of rows also
    final int NO_OF_MEDALS = 3;//the number of columns also
    String[] colleges = {"ITB", "DIT", "ITT", "DKIT"};
    //each data entry/element represents how many medals of the given type a college won
    int [][] data = {
                        {10, 7, 7},
                        {7, 3, 5},
                        {5, 2, 2},
                        {1, 1, 1}
                    };
    //print the header and the table
     System.out.println("        College    Gold  Silver  Bronze   Total");
    //print the name of the college, number of each type of medal, and total medals
    for(int i = 0; i < NUM_COLLEGES; i++){
        //process each row i
        //first we print the name of the college
        System.out.printf("%15s", colleges[i]);
        //then we process actual data ... the number of medals and total
        int total =0;
        //print each row and the update the total variable
        for(int j = 0; j < NO_OF_MEDALS; j++){
            System.out.printf("%8d", data[i][j]);
            total = total + data[i][j];
        }//end inner for
        //print the total of medals for each college and move cursor to next line/row
        System.out.printf("%8d\n",total);
    }//end outer for
}//end main
```

A full program… output

```
ted> TwoDArrayDemo [Java Application] C:\Program Files\Java\jre1.8.0_77\bi
College     Gold  Silver  Bronze   Total
    ITB       10       7       7      24
    DIT        7       3       5      15
    ITT        5       2       2       9
   DKIT        1       1       1       3
```

Asymmetrical 2D arrays

- The number of columns varies from row to row == **row length varies;** in other words, the number of elements in each component array can be different:

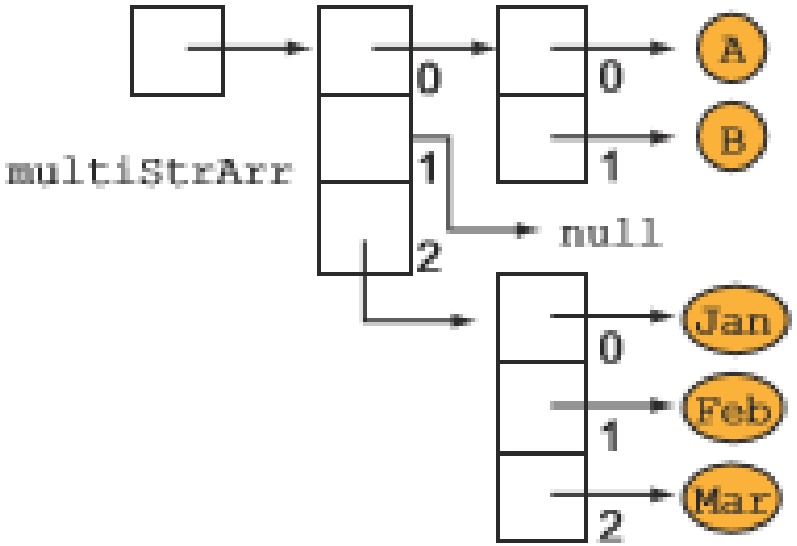
```
int [][] asymData = {
                    {0, 1},        ←——— 2 elements in first row/1D array
                    {34},          ←——— 1 element in second row/1D array
                    {7, 1, 2, 5}   ←——— 3 elements in third row/1D array
                };
// the number of elements in asymData is given by the number of rows/1D arrays: 3
```

- to access all its elements we also use nested loops (can be while, for, enhanced for):

```
for(int[] row: asymData){
    for(int element: row){
        System.out.printf("%5d", element);
    }
    System.out.println();
```

Asymmetrical 2D arrays

```
String multiStrArr[][] = {
        {"A", "B"},
        null,
        {"Jan", "Feb", "Mar"}
    };
```



- Arrays of objects can contain the value ***null;***
- but we must be careful, because we might get a NullPointerException if we try to access the elements of that 1D array, because null cannot have any elements...

System.out.println(a[1][0]); /* will end with the above exception when we try to execute it */

Exercise time…

- **True** or **false?**

- All elements of an array are of the same type. true
- Arrays cannot contain string references as elements. false
- Two-dimensional arrays always have the same number of rows and columns. false
- Elements of different columns in a two-dimensional array can have different data types. false
- A method cannot return a two-dimensional array. false
- A method cannot change the length of an array argument. true