# Fundamentals of Programming 2

**Lecture 7** 

Aurelia Power, TU Dublin - Blanchardstown Campus, 2019

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

#### Sorting Recap

**Sorting** a collection of data is the task of re-arranging that data in a particular order.

- Examples:
- ✓ Sorting numerical data in ascending/descending order;
- ✓ Sorting strings in alphabetical order
- Sorting algorithms covered:
- ✓ Bubble sort
- ✓ Selection sort

#### Sorting Recap – bubble sort

- Inefficient algorithms 

  suitable only for small amounts of data
- It compares each adjacent pair of items in an array/a list and:
- In the case of **sorting ascendingly**, it <u>swaps</u> the items if the first element of the pair is *greater than* the second
- In the case of **sorting descendingly**, it <u>swaps</u> the items if the first element of the pair is *smaller than* the second
- It <u>repeats this process</u> until each element in the array/list is in place (sorted).
- **NOTE:** there can be various implementations of this algorithms

## Sorting Recap – bubble sort

```
int [] array = \{2, 0, 19, 7, 16\};
```

• States of the array when sorting it in increasing order:

$$\{0, 2, 19, 7, 16\} \rightarrow \{0, 2, 7, 19, 16\} \rightarrow \{0, 2, 7, 16, 19\}$$

• States of the array when sorting it in decreasing order:

```
\{2, 19, 0, 7, 16\} \rightarrow \{2, 19, 7, 0, 16\} \rightarrow \{2, 19, 7, 16, 0\} \rightarrow \{19, 2, 7, 16, 0\} \rightarrow \{19, 7, 2, 16, 0\} \rightarrow \{19, 7, 16, 2, 0\} \rightarrow \{19, 16, 7, 2, 0\}
```

#### Sorting Recap – bubble sort descending

#### Sorting Recap – selection sort

- Also inefficient algorithm 

   suitable only for small amounts of data
- It repeatedly selects the :
- ➤ the smallest element of the remaining unsorted portion of the array and moves it to the front of that portion, in the case of sorting ascendingly
- the largest element of the remaining unsorted portion of the array and moves it to the front of that portion, in the case of sorting descendingly
- **Note**: initially, the unsorted portion is the entire array

## Sorting Recap – selection sort

```
int [] array = \{2, 0, 19, 7, 16\};
```

• States of the array **after each swap** when sorting it in <u>increasing order</u>  $\{0, 2, 19, 7, 16\} \rightarrow \{0, 2, 7, 19, 16\} \rightarrow \{0, 2, 7, 16, 19\}$ 

• States of the array **after each swap** when sorting it in <u>decreasing order</u>  $\{19, 2, 0, 7, 16\} \rightarrow \{19, 16, 0, 7, 2\} \rightarrow \{19, 16, 7, 0, 2\} \rightarrow \{19, 16, 7, 2, 0\}$ 

#### Sorting Recap – selection sort

```
for (int i = 0; i < arr.length - 1; i++) {
    int index = i;
    for (int j = i + 1; j < arr.length; j++){
        if (arr[j] > arr[index]) 4
                                              This time we find the
             index = j;
                                              greatest element in the
        }//end if
                                              remaining unsorted
    }//end inner for
                                              portion of the array
    double temp = arr[index];
    arr[index] = arr[i];
    arr[i] = temp;
}//end outer for
```

## Sorting Recap – let's put it all together

```
Class declaration (you can
public class DescendingSortingAlgorithms
                                                                name it anything you like,
     * takes in an array of double values and sorts it in descarding asdit does not
     * it use the selection sort algorithm
                                                                contravene naming rules
     * */
    public static void selectionSortingDescending(double[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {</pre>
            int index = i:
            for (int j = i + 1; j < arr.length; j++) {</pre>
                if (arr[j] > arr[index]){
                    index = j;
                                                                    Method that
                }//end if
            }//end inner for
                                                                     implements the
                                                                    selection sort algorithm
            double temp = arr[index];
                                                                    in decreasing order for
            arr[index] = arr[i];
            arr[i] = temp;
                                                                    double values
        }//end outer for
      //end selectionSortDescending
```

```
/**
 * takes a double array and sorts it from the largest
 * to the smallest using the bubble sort algorithm
public static void bubbleSortDescending(double[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
        for(int j = 0; j < n-i-1; ++j) {
            if(arr[j] < arr[j + 1]) {</pre>
                double temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }//end if
        }//end inner for
    }//end outer for
}//end method
```

Method that implements the bubble sort algorithm to arrange an array of double values in decreasing order

```
/**
  2 bonus overloaded methods to display int arrays
* and double arrays, so we won't have to re-write
* the code for displaying over and over again...
public static void displayArray(int[] arr) {
    for(int a : arr) {
        System.out.print(a + " ");
    System.out.println();
}//end first overloaded
public static void displayArray(double[] arr) {
    for(double a : arr) {
        System.out.print(a + " ");
    System.out.println();
}//end 2nd overloaded
```

Two extra overloaded methods that weren't required in the lab, but it will make the main method more readable since we can call these methods several times there, without having to re-write the code to display the arrays

```
public static void main(String[] args) {
    System.out.println("Using selection sort...");
    double[] arr1 = {3.4, -10, 7.07, -7, 3};
    selectionSortDescending(arr1);
    displayArray(arr1);
    double[] arr2 = {1000, -12, 700.75, 123.1, 2500.3, -5, 43.0};
    selectionSortDescending(arr2);
    displayArray(arr2);
    System.out.println("\nUsing bubble sort...");
    double[] arr3 = {3.4, -10, 7.07, -7, 3};
    bubbleSortDescending(arr3);
    displayArray(arr3);
    double[] arr4 = {1000, -12, 700.75, 123.1, 2500.3, -5, 43.0};
    bubbleSortDescending(arr4);
    displayArray(arr4);
}//end main
```

Main method where we invoke our 2 sorting methods and the second display method

How many times did we call selectionSortDescending?
Twice

How many times did we call bubbleSortDescending?
Twice

Using selection sort...
7.07 3.4 3.0 -7.0 -10.0
2500.3 1000.0 700.75 123.1 43.0 -5.0 -12.0

Using bubble sort...
7.07 3.4 3.0 -7.0 -10.0
2500.3 1000.0 700.75 123.1 43.0 -5.0 -12.0

Sample output ...

How many times did we call displayArray?
4 times...

#### Search Recap

- **Searching** is the task of looking for a particular value in a given data list (such as an array);
- Examples:
- > Searching for a name in a name directory
- > Searching for a value in a data set
- Linear search goes through the array and compares its elements with the search key (the value you are searching for);
- if it finds a match, it will typically <u>return the index</u> in the array where the match was found (it will also exit the loop, and the method, returning that index to the caller);
- if it doesn't find any match after exhausting all the elements, it will <u>return</u> -1 to the caller.
- Linear search is *inefficient* for large amounts of data

```
public class NumericalLinearSeach {
     * @param args; main method to test the methods and run the program
   public static void main(string[] args) {
        int [ ] array = {11, 9, 17, 5, 12, 67, 21, 77, 5};
        System.out.println(linearSearch(array, 12));//4
        System.out.println(linearSearch(array, 1000));//-
        double [ ] array2 = {11.1, 9.5, 17, 5, 12, 67.3, 21, 77, 5.2};
        System.out.println(linearSearch(array2, 67.3));//5
        System.out.println(linearSearch(array2, 100.2));//-
    }//end main method
    /** method that searches the given array of integers for the given key;
     * it returns the index where the key is found (if key is in the array),
    public static int linearSearch(int[] arr, int key) {
        for(int i = 0; i < arr.length; i++) {</pre>
            if(arr[i] == key) {
                return i:
            }//end if
        }//end for
        return -1:
    }//end method
    /** method that searches the given array of doubles for the given key;
     * it returns the index where the key is found (if key is in the array),
     * or else -1 (if key is not in the array)*/
    public static int linearSearch(double[] arr, double key) {
        for(int i = 0; i < arr.length; i++) {</pre>
            if(arr[i] == key) {
                return i;
            }//end if
        }//end for
        return -1;
      /end method
```

#### Class declaration

Main method where we invoke our 2 overloaded methods

Invoke linear search twice, passing it an int array and, first a value that can be found in the array, and, then a value that cannot be found in the array;

Do the same for double array and double values

Method that implements linear search for an int array

We compare 2 ints using the == operator

Method that implements linear search for a double array

We compare 2 doubles using the == operator

# Today

Binary Search

Variable-length arguments lists

• The Arrays class

#### **Binary Search**

- Assume you want to read the chapter on arrays in your java book or search a directory for last names same as Power....
- You wouldn't start at page 1 and leaf through each page until you get to that chapter!!!
- You wouldn't start at A, you would go straight to P.
- When there is some order in the data being searched we can use that to speed things up.
- This is what <u>binary search</u> is based on ...
- The binary search is much more **efficient**, but only works with <u>sorted</u> data

## **Binary Search**

- It cuts the size of the search repeatedly in half; the cutting in half works only because the data is already sorted.
- It locates a value in a sorted list of data by determining whether the value is in the first or second half; it then repeats the same process in that half, and so on, until it finds the value, or else the array can no longer be split into halves.
- Like linear search, the binary search implementation typically <u>returns</u> the index where the value was found, or, if the value wasn't found, it returns -1.

#### Binary Search – the algorithm

```
While the array can be split in half (approx.)
      compare the middle element with the key
             if the middle element is the same as the key
                    return the index of the middle element
             if the middle element is greater than the key
                    update the upper bound of the search (look only at
                    the first half)
             if the middle element is smaller than the key
                    update the lower bound of the search (look only at
                    the second half)
If the array can no longer be split
      return -1
```

#### Binary search - example

int [] arr = {1, 5, 8, 9, 12, 17, 20, 32}; /\* the array is already sorted in increasing order \*/

• We want to see if the value of **15** (also called the **key** or **search key**) is in *arr* 

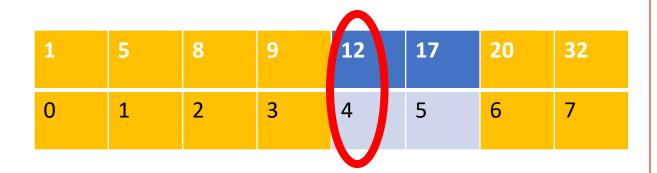
0 1 2 3 4 5 6 7	1	5	8	9	12	17	20	32
	0	1	2	3	4	5	6	7

Let's narrow the search: we split the array in half and compare the last element of the first half (or the middle element): 9 (at index 3) with key; notice that 15 is bigger than 9 so there is no point in looking in that half, since all the elements are smaller than 15 (REMEMBER: the array is sorted, so that's how I know that they are all smaller)

1	5	8	9	12	17	20	32
0	1	2	3	4	5	6	7

We are now looking only at the second half, and split that half also into 2 halves. We compare the last value of the first half: 17 (at index 5) to 15, and notice that it is greater than 15, so our value, if it exists in arr, it must be in this half (so we will not look at the second half of the second half).

#### Binary search - example



We are now looking only at the first half of the second half, and we split this too in two halves( because we only have 2 elements our halves will contain only 1 element each). We compare the last value of the first half: 12 (at index 4) and notice that it is different from 15.

1	5	8	9	12	17	20	32
0	1	2	3	4	5	6	7

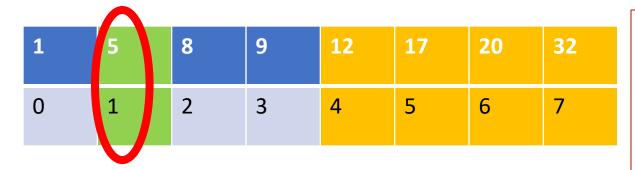
Next, it is trivial to see that it can no longer be split into two, so our search will return -1.

#### Binary search - example

• Let's search the arr, this time, for the value of 5

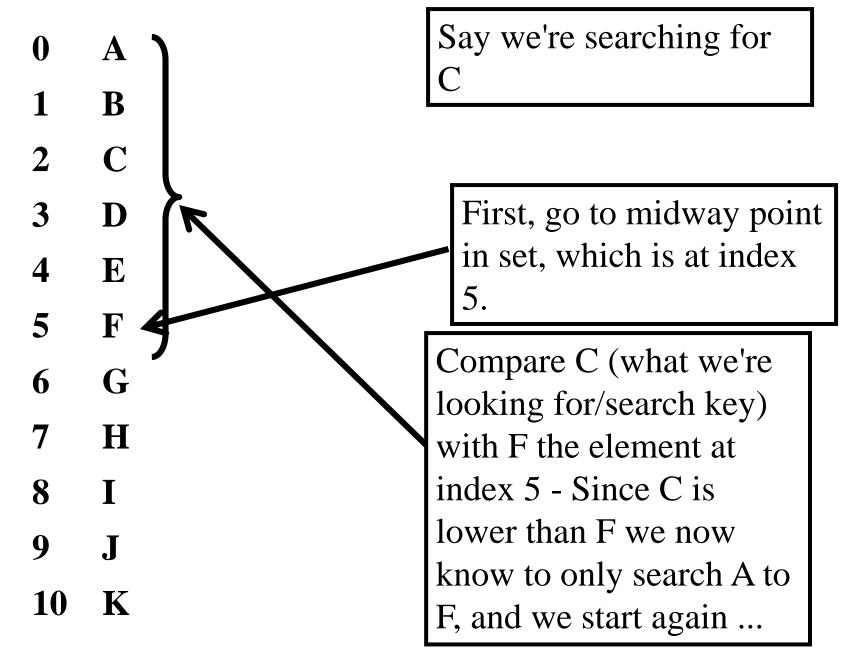
1	5	8	9	12	17	20	32
0	1	2	3	4	5	6	7
			V				

We first split the array in half and compare the last element of the first half is 9 (at index 3) to 5 and notice that 9 is greater, so we will look at the first half only; in other words, if the element does exist in *arr* it must be found in the first half, and discard the second half.

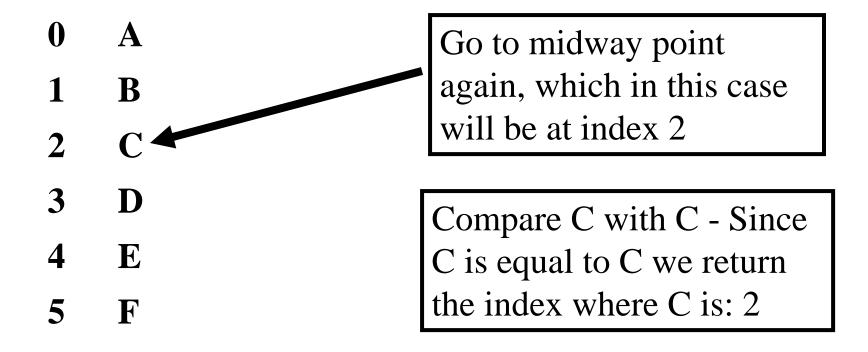


We are now looking only at the first half, and split that half also into 2 halves. We compare the last value of the first half which is 5 (at index 1) and notice that it matches the value that we were looking for, so we return the index where we found the match, in this case 1.

#### Searching ... Chars example



#### **Searching ... Chars example**



#### Binary Search - NOTES

• In our implementation, the **middle element** is found at **index** (length-1)/2; for example:

If length is 7, the middle element is at index (7-1)/2 = 6/2 = 3

If length is 8, the middle element is at index  $(8-1)/2 = 7/2 = 3 \dots \text{ why}???$ 

Because, in java, when we divide an int by another int, it will discard anything that's after the decimal point; so, strictly (mathematically) speaking, 7 divided by 2 should give us 3.5, but, in java, because it's an int divided by an int, it gives another int, and it will automatically discard .5, giving us 3

• The <u>upper and lower bounds are repeatedly updated</u>: How??

if the key is smaller than mid-element, we update the upper bound to the value of mid-element index -1; if the key is greater than mid-element, we update the lower bound to the value of mid-element index + 1.

- The index of the middle element is also repeatedly updated, based on which half we search;
- If the size of the array (or size of subsequent so-called *halves*) is **odd**, then one of the halves must contain one more element than the other.

#### Binary search ... quiz

• Given the following array: {1, 7, 12, 21, 57}, what is the index of the middle element using the implementation shown on previous slides?

$$(5-1)/2 = 2$$

- Given the following array: {1, 77, 3, 21, 16}, and the key search 77, what will the binary search return?
- 1 ... why???

Because the array is supposed to be sorted for the binary search to work, but in this case is not, so it will look in the second half (which made up of 21 and 16) which does not have 77, because element at index 2 (which is 3) is smaller than 77.

```
Method that implements the binary
public class BinarvSearchDemo {
    public static int binarySearch(int[] data, int key) {
                                                                   search for an int array
         int low = 0;
                                                                    Initialise the search boundaries to the end
         int high = data.length - 1;
                                                                    indices of the array data
         while (high >= low) {
                                                                    Compute the middle index
             int middle = (low + high) / 2;
             if(data[middle] == key) {
                                                                    If the element at middle index matches the key,
                 return middle;
                                                                    return the middle index
             }//end if
             if (data[middle] < key) {</pre>
                                                                    If the element at middle index is smaller than
                 low = middle + 1;
                                                                    the key, update the lower bound so we will look
             }//end if
             if(data[middle] > key)
                                                                    only at the second half
                 high = middle - 1;
                                                                   If the element at middle index is greater than
             }//end if
                                                                   the key, update the upper bound, so we will
        }//end while
        return -1;
                                                                   look only at the first half
     //end method binary search
                                                  If we no longer can split the halves into further halves, return -1, because
    /**
                                                 that value does not exist in the array
     * @param args; main method to test the binarySearch method
    public static void main(String[] args) {
                                                                                                 Main method where
        int [ ] arr = {1, 5, 8, 9, 12, 17, 20, 32};
                                                                                                 we invoke the
        int a = binarySearch(arr, 15);
                                                                                                 method twice
        System.out.println((a != -1) ? ("Key found at index " + a) : "key not found");
        a = binarySearch(arr, 5);
        System.out.println((a != -1) ? ("Key found at index " + a) : "key not found");
    }//end main method
}//end class
```

#### Variable-length Argument Lists

- Allows us to create methods for which the parameter list varies;
- The syntax is:

public static returnType methodName ( dataType... variableName){}

- Note the ellipsis (...) which follows the dataType
- When we invoke these type methods we can pass 1 argument, or 2 arguments or 3, and so on...

#### **Example:**

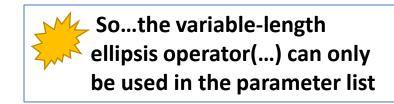
- Defining the method

```
public static int addIntegers (int... integers){ /*code here to add ints, make sure last statement is a return statement*/ }
```

- Calling the method (for instance, in the main method)

```
int total1 = addIntegers (2, 1); // yields 3
int total2 = addIntegers (3, 4, 0); // yields 7
int total3 = addIntegers (3, 4, 2, 1); // yields 10
```





- There are some restrictions though:
- 1. You can only place the variable argument at <a href="the-end of the parameter list">the end of the parameter list</a> public static void displayIntegers (int ... integers, String str){/\* some code \*/} /\* will NOT compile \*/
  public static void displayIntegers (String str, int ... integers){/\* some code\*/} /\* will compile \*/
- 2. Following from the above rule, you can <u>only have 1 variable-length argument in a given method</u> public static int addIntegers (int ... integers1, int ... integers2){ }
- /\* will also NOT compile \*/
- 3. You cannot return a variable length array from a method: public static int... someMethodName(int... ints){} // will NOT compile
- 4. You cannot declare a variable that has as datatype a variable length array:

int... ints; // will not compile

String... names; // will also NOT compile

#### Variable-length Argument Lists

• Java treats the variable-length argument list as an array of the specified type; for instance, in the method definition below:

#### public static void displayIntegers (int ... integers){ }

<u>int...</u> integers can be regarded as <u>int [] integers</u> for which the length is decided when we call the method, depending on how many arguments we passed in the call;

- if we call the method with 2 ints, then the length of the array *integers* is 2
- if we call it with 1 int, then the length of the array integers is 1
- if we call it with 100 ints, then the length of the array *integers* is ... 100

#### Variable-length Argument Lists – in class exercises

```
    Implement the method addIntegers

public static int addIntegers (int... integers){
    int total = 0;
    for(int i = 0; i < integers.length; i++){
        total += integers[i];
    return total;

    Let's call addIntegers (for instance in the main) and see what gives us:

System.out.println(addIntegers(3)); //prints 3
System.out.println(addIntegers(3, 4, "hello")); /* does not compile because you cannot mix the type of arguments; we can only pass ints */
System.out.println(addIntegers(3, 4, 2, 1)); // prints 10
System.out.println(addIntegers()); //Prints 0
```

#### The **Arrays** class

- It is located in the **java.util** package (so you need to import it).
- is an **utility class** that contains all sorts of static methods that help us work with and manipulate arrays by simply invoking its methods:
- 1. Overloaded methods to sort arrays of various types
- 2. Overloaded methods to search arrays of various types
- 3. Overloaded methods that allows us to fill arrays of various types
- 4. Overloaded methods to <u>compare arrays for equality</u> (again for various types)
- 5. Overloaded methods to copy arrays
- 6. And many other methods ... check the API

#### The **Arrays** class

```
    Let's consider the following 2 arrays:

int [] array1 = \{16, 5, 12, 77, 5\};
double [] array2 = {11.1, 9.5, 5, 77.2, 5.2};
- We can sort them (default order is ascending order):
       Arrays.sort(array1);
       Arrays.sort(array2);
- We can print their contents without writing loops:
System.out.println(Arrays.toString(array1));
// [5, 5, 12, 16, 77]
System.out.println(Arrays.toString(array2));
// [5.0, 5.2, 9.5, 11.1, 77.2]
```

#### The **Arrays** class

```
-We can <u>search them</u> (using binary search logic):
System.out.println(Arrays.binarySearch(array1, 77)); //8
System.out.println(Arrays.binarySearch(array2, 67.3)); //7
-We can <u>compare them for equality</u> to other arrays:
int [] array1 = \{16, 5, 12, 77, 5\};
double [] array2 = {11.1, 9.5, 17};
int [] array3 = {16, 5, 12, 77, 5};
double [] array4 = {11.1, 9.5, 17};
Arrays.sort(array2);
System.out.println(Arrays.equals(array1, array3));
//true
System.out.println(Arrays.equals(array2, array4));
//false
You cannot compare array arrays of different types: System.out.println(Arrays.equals(array1, array2)); // will not compile because array1 stores ints, and array2 stores doubles
```

#### Arrays class - exercises

```
    Consider the following code and tell what will output:

int [] array1 = \{2, 3, -1, 0, 5\};
double [] array2 = {67.3, 21, 77, 5.2};
int [] array3 = \{2, 3, -1, 0, 5\};
double [] array4 = \{11, 9.0, 3.2\};
Arrays.sort(array1);
System.out.println(Arrays.equals(array1, array3));
// false
Arrays.sort(array3);
System.out.println(Arrays.equals(array2, array4));
// false
System.out.println(Arrays.equals(array3, array1));
// true
System.out.println(Arrays.equals(array3, array4));
// will not compile because array3 stores ints, array4 stores doubles... so you cannot compare them
```