MACQUARIE University

Faculty of Science and Engineering

COMP125 Fundamentals of Computer Science
Workshop Week 6

Learning outcomes

By the end of this session, you will know the basics of objects and testing in Java. In particular, you will be able to,

a. Perform binary search on an array;

b. Operate on array of objects;

c. Create array of objects

Import project `workshop06template` from `workshop06template.zip`.

1. **Apply binary search on an array**

Consider the following **pseudo-code** is being used to perform binary search.

```
Input:
1. array arr: assumed to be sorted in ascending order
2. target (of the type as each item of the array arrr)

Output:
index in array at which target is found. -1 if not found

Process:

//Search space: from first to last item

first = 0
last = arr.length - 1
while(first <= last)
   median = (first+last)/2
   if target is equal to arr[median]
      OUTPUT median

   //options left target < arr[median] or target > arr[median]

   if target < arr[median] //if present, it's in the left half
      last = median - 1
   else //target > arr[median] means if present, it's in the right half
      first = median + 1
//loop ends means first > last and search space exhausted
OUTPUT - 1
```

For the array

```
{1, 4, 5, 7, 9, 23, 47, 50, 58, 58, 58, 58, 88, 90, 95, 99}
```

Trace the execution of the above algorithm for the following targets,

- 58

first	last	median	arr[median]

- 50

first	last	median	arr[median]

- 47

first	last	median	arr[median]

- 59

first	last	median	arr[median]

2. **Variations of binary search**

   a. Applying binary search on integer array sorted in descending order

   b. Applying binary search on array of objects. You cannot compare two objects of a class, say `Rectangle`, as,

```
Random rand = new Random();
Rectangle r1 = new Rectangle(rand.nextInt(4), rand.nextInt(5));
Rectangle r2 = new Rectangle(rand.nextInt(4), rand.nextInt(5));
if(r1 > r2) //INCORRECT
        System.out.println(``first rectangle is bigger'');
else
        System.out.println(``second rectangle is bigger'');
```

   However, `compareTo`, if defined should logically return a positive value (typically 1), if the calling object is *more than* the object passed as parameter.

```
public class Rectangle {
        private int width, height;

```

```
        //getters, setters, and constructors

        public int area() {
                return width * height;
        }

        //comparison based on area
        public int compareTo(Rectangle other) {
                if(this.area() > other.area())
                        return 1;
                if(this.area() < other.area())
                        return -1;
                return 0;
        }
}
```

Thus, we should replace the comparison statement by `r1.compareTo(r2) > 0`.

```
Random rand = new Random();
Rectangle r1 = new Rectangle(rand.nextInt(4), rand.nextInt(5));
Rectangle r2 = new Rectangle(rand.nextInt(4), rand.nextInt(5));
if(r1.compareTo(r2)) //logically, r1 > r2 based on area
        System.out.println(``first rectangle is bigger'');
else
        System.out.println(``second rectangle is bigger'');
```

Similarly,

- `(r1 < r2)` → `(r1.compareTo(r2) < 0)`, and,
- `(r1 == r2)` → `(r1.compareTo(r2) == 0)`

3. **Implement search algorithms**

The project `workshop06template` contains a *service* class `ArrayService`.All methods of this class are `static` and operate on the array passed to them. Complete the following `static` methods in this class,

a. `linearSearch`: Perform linear search for an item on the array. The parameters are the array and item to be searched. The method should return the index at which the first occurrence of item is found, and -1 if not found. Write your unit tests to ensure that the method is correct. (Hint. Linear search was covered in lectures back in Week 3.)

b. `binarySearch`: Perform binary search for an item on the array assumed to be sorted in ascending order. The parameters are the array and item to be searched. The method returns the index at which the item is found (not necessarily the index of the first occurrence because binary search does not scan the array sequentially), and -1 if not found. Write your unit tests to ensure that the method is correct. (Hint. Binary search was covered in lectures in Weeks 5 and 6.)

c. `search`: Write a method that when passed an array and an item to be searched, calls either the binary search method or the linear search method, depending on whether the array is sorted in ascending order or not. Note that you can use other methods of `ArrayService` – for example the `isSorted` method might be useful. Write your unit tests to ensure that the method is correct.

Discuss in pairs the best and worst case time complexities of each of the three algorithms, and comment on the convenience or otherwise of implementing the third function.

Solution: Refer to `ArrayService.java`

4. **Understanding the process of creating array of objects**

Take a look at the document `arrayOfObjectsSummary.pdf` (included in the `eclipse project` that you imported) and repeat the process illustrated in it for the class `Trip` and the client `TripListClient` that creates an array of `Trip` objects and operates on it.

5. **Create an array of objects**

You are provided with a completed `StockItem` class. Complete the code in `StockItemListClient` that creates and operates on an array of `StockItem` objects, similar to the code in `TripListClient` that creates and operates on an array of `Trip` objects. You can use the following configuration values,

- array should be of size 5
- `name` of first item should be "Item 1", of second item should be "Item 2" and so on. In general, `name` of item at index `i` should be `"Item "+(i+1)`.
- `unitPrice` of item at index `i` should be `2*i + 2`. (you can assign an integer to a double variable so it's all good)
- `quantity` of item at index `i` should be `2*i - 8`.

Display the item with the maximum `totalStockPrice()` as defined in the `StockItem` class.

6. **Weekly Task Week 6 (assessed)**

Complete the following methods based on the specifications,

a. **existsInFirstHalf**: returns `true` if `target` exists in the first half of the array `arr`, and `false` otherwise. If the array contains an odd number of items ($2*k+1$, where k is an integer), then the first half contains the first k items.

```
public static boolean existsInFirstHalf(double[] arr, double target);
```

```
public static boolean existsInFirstHalf(double[] arr, double target)
    {
        if(arr == null)
                return false;
        int end = arr.length/2;
        for(int i=0; i < end; i++) {
                if(arr[i] == target) {
                        return true;
                }
        }
        return false;
}
```

b. **getCount**: returns the number of times `target` exists in `arr`.

```
public static int getCount(double[] arr, double target);
```

```
public static int getCount(double[] arr, double target) {
        if(arr == null)
                return 0;
        int count = 0;
        for(int i=0; i < arr.length; i++) {
                if(arr[i] == target) {
                        count++;
                }
        }
        return count;
}
```

The time complexity in the best case is $O(log_2(n))$ and in the worst or average case is $O((n)$.

c. **getMostFrequentItem**: Assuming the method `getCount` is completed and available, returns the item that occurs the most number of times in `arr`. If there is a tie, the candidate that occurs first in the array is returned. For example, the method returns 8 for the array $\{8,8,8,1,1,1,6,6,6\}$. Return 0 if the array is empty or null.

```
public static int getMostFrequentItem(double[] arr);
```

```
public static int getMostFrequentItem(double[] arr) {
        if(arr == null || arr.length == 0) {
                return 0;
        int freqIndex = 0;
        for(int i=1; i < arr.length; i++) {
                if(getCount(arr, arr[i]) > getCount(arr, arr[
                    freqIndex)) {
                        freqIndex = i;
                }
        }
        return freqIndex;
}
```

d. (students volunteer for this part) **getFirstIndex**: Consider the following method for binary search.

```
public static int binarySearch(double[] arr, double target) {
        int first = 0;
        int last = arr.length - 1;
        while(first <= last) {
                int median = (first+last)/2;
                if(target==arr[median])
                        return median;

                if(target > arr[median])
                        first = median + 1;
                else
                   last = median - 1;
        }
        return -1;
}
```

The above method returns index 3 while searching for the item 24 in the array $\{0,24,24,24,24,24,24\}$. Modify the method so that it returns the **first** index of the item from the array (should return 1 for the above example). What is the time complexity for your algorithm?

```
public static int binarySearchV2(double[] arr, double target) {
        int first = 0;
        int last = arr.length - 1;
        int median = -1;
        while(first <= last) {
                median = (first+last)/2;

                if(target == arr[median])
                        break; //exit the loop

                if(target > arr[median])
                        first = median + 1;
                else
                   last = median - 1;
        }
        if(median == -1)
                return -1;

        while(median >= 0 && target == arr[median]) {
                median--;
        }

        return median + 1;
}
```