EXERCISE 1: Comparables & **Comparators** (live-coding)

Download **`precept3.zip`** from the precepts page, unzip the project and open it using IntelliJ. Follow along with your preceptor (using the annotated code in the next page) to do the following:

(a) Define a *natural (default)* comparison behavior for the **`Point2D`** class and use it in a simple test client. Proceed according to the following steps:

- Modify the class declaration of **`Point2D`** to make it implement the **`Comparable`** interface.
- Implement the **`compareTo`** method. This method allows the point to be compared to another given point (passed as an argument to the method).
    - Use the *y-coordinate* for comparison and break ties using the *x-coordinate*.
    - Return **`1`** if the point is greater than the method argument, **`-1`** if it is less and **`0`** otherwise.
- Complete the given test program to sort the array according to the natural order defined in the **`compareTo`** method.

(b) Define an *alternate* comparison behavior for 2D points and use it in a simple test client. Proceed according to the following steps:

- Uncomment the code marked as **`/* *** PART (B) *** */.`**
- Complete the implementation of class **`DistanceToOrder`** such that it allows comparing between two given points based on their distance to a given reference point.
    - Make the class implement the **`Comparator`** interface.
    - Implement the constructor to receive and store the reference point.
    - Implement the **`compare`** method. This method compares the two given argument points: Returns **`1`** if the first argument is farther from the reference point than the second argument, **`-1`** if it is closer and **`0`** otherwise.
- Complete the given test program to sort the array according to the distance of the points from the origin **`(0, 0)`**.

```
1  public class Point2D implements Comparable<Point2D> {
2
3      private final double x, y;
4
5      public Point2D(double x, double y) {
6          this.x = x;
7          this.y = y;
8      }
9
10
11     // Returns the square of the Euclidean distance between two points.
12     public static double distanceSquared(Point2D p, Point2D q) { … }
13
14     // Compares by y-coordinate, breaking ties by x-coordinate.
15     public int compareTo(Point2D other) {
16         if (this.y < other.y) return -1;
17         if (this.y > other.y) return +1;
18         return Double.compare(this.x, other.x);
19     }
20
21     // Returns a Comparator for comparing by distance to a reference point.
22     public static Comparator<Point2D> distanceToOrder(Point2D ref) {
23         return new DistanceToOrder(ref);
24     }
25
26     // compare points according to their distance to a reference point.
27     private static class DistanceToOrder implements Comparator<Point2D> {
28         private Point2D ref;
29
30         private DistanceToOrder(Point2D ref) {
31            this.ref = ref;
32         }
33
34         public int compare(Point2D p, Point2D q) {
35             double dist1 = distanceSquared(p, ref);
36             double dist2 = distanceSquared(q, ref);
37             return Double.compare(dist1, dist2);
38         }
39     }
40
41     public static void main(String[] args) {
42         int n = Integer.parseInt(args[0]);
43
44         Point2D[] array = new Point2D[n];
45         // … fill the array with random points.
46
47         Arrays.sort(array);
48         // … print the array
49
50         Comparator<Point2D> cmp = Point2D.distanceToOrder(new Point2D(0, 0));
51         Arrays.sort(array, cmp);
52         // … print the array
53 }
```

1 *Promise to have method* `compareTo` *that allows comparing this point to other points.*

2 *Required by the* `Comparable` *interface*

Defines a ***natural (default) order*** *between points.*

A *Provides access to an an instance of the* `Comparator`

B *Promise to implement method* `compare` *that allows comparing two different points.*

C *Required by the* `Comparator` *interface.*

Defines an ***alternate order*** *between points.*

3 *Uses the* ***natural order****. Requires the array elements to be of a type that implements* **`Comparable`**.

D *Uses the* ***alternate order****. Requires an instance of the* **`Comparator`**.

EXERCISE 2: Three-Way Merge Sort

3-way Merge sort is a variant of the Merge sort algorithm that considers 3 “equal” subarrays instead of 2 subarrays.

(a) Given 3 sorted subarrays of size $\frac{n}{3}$, how many comparisons are needed (in the worst case) to *merge* them to a sorted array of size n? Provide your answer in tilde notation.

(b) What is the *order of growth* of the number of compares in 3-way Merge Sort as a function of the array size n?

(c) Given a choice, would you choose 3-way or 2-way merge sort? Justify your answer.

EXERCISE 3: Algorithm Design

Let $a = a_0, a_1, \ldots, a_{n-1}$ be an array of length n. An array b is a circular shift of a if it consists of the subarray $a_k, a_{k+1}, \ldots, a_{n-1}$ followed by the subarray $a_0, a_1, \ldots, a_{k-1}$ for some integer k. In the example below, b is a circular shift of a (with $k = 7$ and $n = 10$).

sorted array a[]

1	2	3	5	6	8	9	34	55	89

circular shift b[]

34	55	89	1	2	3	5	6	8	9

Suppose that you are given an array b that is a circular shift of some sorted array (but you have access to neither k nor the sorted array). Assume that the array b consists of n comparable keys, no two of which are equal. Design an efficient algorithm to determine whether a given key appears in the array b. Your algorithm should run in $O(\log n)$.

ASSIGNMENT TIPS: Autocomplete

(1) Given an array of elements with duplicates, can we use the book implementation of Binary Search to find the ***first occurrence*** of an element?

- The standard implementation of Binary Search finds *an* occurrence, which is not necessarily the *first* occurence.
- Finding *an* occurrence and then scanning left to find the first occurence yields a linear running time (in the worst case), which is not good!
- In this assignment, you will have to modify Binary Search to find the first (and last) occurence of an element in a sorted array in logarithmic time (in the worst case).
- For full credit, your algorithm has to make at most $1 + \lceil log_2 n \rceil$ compares. However, if your algorithm in $O(\log n)$ but makes more than $1 + \lceil log_2 n \rceil$ compares, you will lose *only* 1 point.

(2) What is the order of growth of the **`substring`** method?

- Creating a substring of length r takes time proportional to r.
- Note that the string comparison functions in the assignment should take time proportional to the number of characters needed to resolve the comparison.

  Example: The comparison between `X="AAAAAAA"` and `Y="AABBB"` can be resolved when the first `"B"` in `Y` is reached. The comparison function should not take time proportional to the size of `X` or the size of `Y`. It should take time proportional to the number of characters needed to resolve the comparison!
- Most uses of the **`substring`** method in the **`compare`** functions do not meet the above time constraint. So, be careful!

(3) Are there other things I should note about the assignment?

Definitely! This is why we have a Checklist for the assignment. You might want to check the answers for the following questions in the Checklist:

- What is meant by an *immutable* data type?
- What is the meaning of the type parameter `Key` in the following function declaration?

```
public static <Key> int firstIndexOf(Key[] a, Key key,
                                     Comparator<Key> comparator)
```

- What's a good way to get a `Comparator` object to use for *testing*?

(4) A video that provides some tips for the assignment is available on the assignment Checklist page. The video was made in 2014, so a few things are outdated, but most of it is still useful (for example, the API has changed).