EECS2030 (Section E) Summer 2025 Review Notes on OOP Tracing Point, PointCollector, and PointCollectorTester

CHEN-WEI WANG

Contents

1	Class Point	2
2	Class PointCollector	3
	2.1 Attributes of PointCollector	3
	2.2 Constructor of PointCollector	4
	2.3 Mutator Method addPoint of PointCollector	5
	2.4 Accessor Method getPointsInQuadrantI of PointCollector	6
3	Class PointCollectorTester	8

1 Class **Point**

```
1
    class Point {
\mathbf{2}
      private double x;
3
      private double y;
      public Point(double x, double y) {
4
5
       this.x = x;
6
       this.y = y;
 7
      }
8
      public double getX() { return this.x; }
9
      public double getY() { return this.y; }
10
   }
```

Figure 1: Template for Two-Dimensional Points

• Figure 1 (page 2) shows the complete definition of class **Point**, which is a template for entities (i.e., points) on a two-dimensional plane. The common attributes (Lines 2 and 3) of all points are the x-coordinate and y-coordinate:

2 private double x; 3 private double y;

• To create an instance of the above **Point** class, you need to make a call to its (in this case, the only) constructor (Lines 4 to 7) by giving two argument values whose types match the types of the constructor's two parameters.

```
4 public Point(double x, double y) {
5 this.x = x;
6 this.y = y;
7 }
```

For example, the call Point(3, 4) is valid, because the first argument value 3 matches the type of the first parameter x, and the second argument value should 4 matches the type of the second parameter y.

There are occurrences of name clashes of variables in the above constructor: input parameters \mathbf{x} and \mathbf{y} clash with, respectively, attributes \mathbf{x} and \mathbf{y} .

- There are no compilation errors, but both sources (RHS) of the variable assignments at Line 5 and Line 6 (i.e., x and y) reference the input parameters x and y (rather than the attributes). This phenomenon is called variable shadowing.
- The use of "this." preceding both targets (LHS) of the variable assignments (Line 5 and Line 6) are *necessary* for disambiguating references to the attributes **x** and **y**.
- To retrieve values of the two private attributes **x** and **y** of a **Point** object, one must invoke the two public accessor methods:

```
8 public double getX() { return this.x; }
9 public double getY() { return this.y; }
```

2 Class PointCollector

```
1
    public class PointCollector {
 2
     private Point[] points;
 3
     private int nop; /* number of points */
 4
     public PointCollector() {
 5
       this.points = new Point[100];
 6
     }
 7
     public void addPoint(double x, double y) {
 8
       this.points[this.nop] = new Point(x, y);
9
       this.nop ++;
10
     }
11
     public Point[] getPointsInQuadrantI() {
12
       Point[] ps = new Point[this.nop];
13
       int count = 0; /* number of points in Quadrant I */
14
       for(int i = 0; i < this.nop; i ++) {</pre>
15
         Point p = points[i];
16
         if(p.getX() > 0 && p.getY() > 0) {
17
           ps[count] = p;
18
           count ++;
19
         }
20
       }
21
       Point[] q1Points = new Point[count];
22
       /* ps contains null if count < nop */
23
       for(int i = 0; i < count; i ++) {
24
         q1Points[i] = ps[i]
25
       }
26
       return q1Points;
27
     }
28
     public int getNumberOfPoints() {
29
       return this.nop;
30
     }
31
   }
```

Figure 2: Template for Entities which Collect Points

2.1 Attributes of PointCollector

Figure 2 (page 3) shows the complete definition of class **PointCollector**, which is a template for entities (i.e., collectors of points), each of which collecting a list of points on the two-dimensional plane. The common attributes (Lines 2 and 3) of all point collectors are:

- Line 2: An array of points, which is of type Point[]. Declaring the attribute

private Point[] points;

means that at runtime, each point collector has an array named **points**, where each slot stores the address (which may be **null**) of some **Point** object.

- Line 3: An integer counter named **nop** (standing for <u>n</u>umber <u>of</u> points), which serves two purposes:

- 1. Records how many points have been collected (i.e., added to the **points** array) so far. Notice that the value of **nop** should start with **0**, and get incremented by one as each new **Point** object is collected, until it reaches **points.length**.
- 2. Indicates where in points (i.e., an index value between 0 and points.length 1). When the value of nop reaches its maximum points.length, meaning that all slots of the points array are occupied by addresses of Point objects, this maximum value also means that storing a next Point object is not possible (because the value points.length is out of the index bounds for array points).
- Lines 28 30: To retrieve values of the private attribute nop of a PointCollector object, one must invoke the public accessor method:

```
28 public int getNumberOfPoints() {
29 return this.nop;
30 }
```

2.2 Constructor of PointCollector

To create an instance of the above **PointCollector** class, you need to make a call to its (only) constructor (Lines 4 to 6) by giving zero argument values. For example, consider the the following line

PointCollector pc = new PointCollector();

where

Step 1:

The RHS (i.e., new PointCollector()) creates a new object of type PointCollector.

Step 2:

The LHS (i.e., PointCollector pc) declares a variable pc which, at runtime, stores only an address of some PointCollector object.

Step 3:

The middle equal sign (=) assigns (i.e., stores) the address of the new PointCollector object (created in Step 1) to the PointCollector address placeholder (declared in Step 2).

Here is a visual summary of the effect of executing the above three steps (where we use an addition dashed arrow to make it explicit that the value of **nop** can be seen as pointing to a position in array **pc.points**):



where

- The path pc.nop (where pc is the *context object*) brings us to the value of attribute nop of the specific PointCollector instance pc.
- The path pc.ponts (where pc is the context object) brings us to the value of attribute points. That is, pc.points denotes the address of an array object, whose each slot stores the address of some Point object.

Why is it the case that pc.nop has the value **0** and pc.points has the value of an array whose each slot stores the null (i.e., unknown address) value? Inspect the definition of the constructor in class PointCollector (Lines 5 to 6 in Figure 2, page 3):

```
4 public PointCollector() {
5 this.points = new Point[100];
6 }
```

- When each instance of type PointCollector is created, it is critical to initialize the points array that it possesses (so that subsequent method calls on each PointCollector object can access/modify its points array without triggering a NullPointerException). That is, here is a wrong implementation of the constructor of PointCollector:

```
public PointCollector() { /* wrong implementation of the constructor */
    /* points array is implicitly initialized to null */
}
```

where the **points** array is not explicitly initialized to certain length, so it is implicitly assigned to the default value of reference types: **null**. Here is a visual summary of the effect of the above wrong implementation of constructor:



Why is it wrong? Try access contents of the points array, for example, calling pc.points[0], then it will trigger a NullPointerException, because pc.points is null and calling pc.points[0] is equivalent to calling null.points[0], which simply does not make sense.

- Line 5 initializes points as an array of size 100 (just for this example as the assumed maximum number of points that each PointCollectr may collect). We only initializes the array points with a fixed size, without storing in any of its slots addresses of Point (simply because initially no points have been collected yet), so all slots in points store values null only.
- Since there is no explicit assignments that given an initial value to the attribute nop, it is initialized (implicitly) with the default value of integers: 0.

Remark: The above digram shows a sensible configuration of objects: 1) pc.points.length is 100, meaning that we can in the future store at most 100 objects of type Point into it; and 2) pc.nop is 0, meaning that so far we have collected zero points, and that the next (i.e., very first) point object to be stored will be stored at pc.points[0].

2.3 Mutator Method addPoint of PointCollector

The mutator method void addPoint(double x, double y) (Lines 7 to 10, page 3) defines how a new point with coordinate values x and y can be added to the point collector.

```
7 public void addPoint(double x, double y) {
8 this.points[this.nop] = new Point(x, y); /* store the new point at index 'nop' of 'points' */
9 this.nop++; /* increment # of points collected, update index for storing next point */
10 }
```

There are two steps involved:

Step 1 (Line 8):

8

```
this.points[this.nop] = new Point(x, y);
```

Step 1.1: The RHS (i.e., **new Point(x, y))** creates a new **Point** object, by calling the constructor of **Point**: the parameter values x and y of method addPoint are used as argument values for the call to the constructor call **Point(x, y)**.

Step 1.2: The LHS (i.e., this.points[this.nop]) brings us to the specific slot at index nop in array points.

Step 1.3: The middle equal sign (=) assigns/stores the address of the new Point object (created in **Step 1.1**) at index **nop** in array **points**. Remember that one of the purposes of counter **nop** is that it indicates where a new **Point** to be collected should be stored.

Step 2 (Line 9):

9 **this**.nop++;

Now that from **Step 1** we already store a new **Point** object at index **nop** of array **points**, this line updates (by incrementing) the value of **nop** so that it still serves its purpose well. For example, if in **Step 1** the value of **nop** is 2, that means so far the slots **points[0]**, **points[1]**, **points[2]** are all occupied by addresses of some **Point** object. So to make sure that

- value of nop correctly records the number of Point objects collected so far (which should be 3 rather than 2), and that
- value of nop indicates the position for storing the next new Point to be collected (which should be points[3] rather than points[2]),

we increment the value of **nop** by **1**.

2.4 Accessor Method getPointsInQuadrantI of PointCollector

The accessor method **Point[] getPointsInQuadrantI()** (Lines 11 to 27, page 3) defines, out of the **Point** objects collected so far, how to return an array containing only those points which are located at the first quadrant (i.e., those points whose both coordinates are positive).

First of all, we need to understand that simply returning points (which is also of type Point[]) is not acceptable, because: 1) if nop < points.length, there are points.length - nop slots storing null values; and 2) even if nop == points.length, not necessarily all points collected so far are all located in the first quadrant. Therefore, this accessor method getPointsInQuadrantI is supposed to return an array of Point objects, whose length corresponds to exactly the number of collected points that are located in the first quadrant.

There are three steps involved:

Step 1 (Lines 12 to 20):

```
12
   Point[] ps = new Point[this.nop];
   int count = 0; /* number of points in Quadrant I (Q1) */
13
14
   for(int i = 0; i < this.nop; i ++) {
15
     Point p = this.points[i]; /* store the current point under consideration into p */
     if(p.getX() > 0 \& p.getY() > 0) \{ /* the point being considered is in 1st quadrant */
16
17
       ps[count] = p; /* copy what's stored in 'p' to slot at index 'count' of 'ps' */
18
       count ++; /* increment # of Q1 points, update index for storing next Q1 point */
19
     }
20
   }
```

- Line 12 initializes an array of size nop, indicating that might be up to nop points that we have collected are located in the first quadrant. Why? Given that so far we have collected nop (rather than points.length) Point objects in the point collector, the maximum number of Point that are located in the first quadrant is simply nop (i.e., when all points collected are located in the first quadrant).

- As we iterate through the points array (slots points[0], points[1], ..., points[nop - 1]) some collected points might be located in the first quadrant, and some might not be. As a result, we need a separate counter count that records exactly how many points are located in the first quadrant. How count is used (Lines 17 and 18, Figure 2, page 3) to keep track of how many collected Point objects are located in the first quadrant is analogous to how nop is used (Lines 8 and 9, Figure 2, page 3) to keep track of how many Point objects are collected so far.

Step 2 (Lines 21 to 25):

```
21 Point[] q1Points = new Point[count];
22 /* ps contains null if count < nop */
23 for(int i = 0; i < count; i ++) {
24  q1Points[i] = ps[i]
25 }
```

From Step 1, we have updated the integer variable count so that it records the number of collected points that are located in the first quadrant, and that the count slots ps[0], ps[1], ..., ps[count - 1] store addresses of such first-quadrant points. Notice that it is the case that count <= nop:

Case 1) if **count** == **nop**, then all slots of array **ps** (whose length is **nop** as can be seen in Line 12 in Figure 2, page 3) are occupied;

Case 2) if **count** < **nop**, then at <u>at least one</u> slots of array **ps** (i.e., **ps[count]**, **ps[count** + **1]**, ..., **ps[nop]**) store the **null** points (simply because <u>not all</u> points collected happen to be in the first quadrant).

If the above **Case 2** is true, meaning that there are **null** values in array **ps**, then we cannot simply return **ps** as the return value for method **getPointsInQuadrantI**.

- Therefore:
 - * In Line 21 we initialize an array q1Points whose length corresponds the exact number of first-quadrant points (i.e., the value of count).
 - * In Lines 23 to 25, we copy addresses of all such first-quadrant (stored in ps[0], ps[1], ..., ps[count - 1]) to their corresponding positions in array q1Points to be returned: q1Points[0], q1Points[1], ..., q1Points[count - 1].

Step 3 (Line 26):

26 | **return** q1Points;

Now that the previous two steps store addresses of all first-quadrant points (no more, and no less) into the array q1Points, we return it as the return value of method getPointsInQuadrantI (whose return type is Point[]).

3 Class PointCollectorTester

After understanding the descriptions of the attributes and methods of the classes **Point** (Section 1) and **PointCollector** (Section 2) from the previous two sections, let us learn how objects of these two classes may interact.

```
1
   public class PointCollectorTester {
2
     public static void main(String[] args) {
3
       PointCollector pc = new PointCollector();
       System.out.println(pc.getNumberOfPoints()); /* 0 */
4
5
       pc.addPoint(3, 4);
 6
       System.out.println(pc.getNumberOfPoints()); /* 1 */
 7
       pc.addPoint(-3, 4);
       System.out.println(pc.getNumberOfPoints()); /* 2 */
 8
       pc.addPoint(-3, -4);
9
10
       System.out.println(pc.getNumberOfPoints()); /* 3 */
11
       pc.addPoint(3, -4);
       System.out.println(pc.getNumberOfPoints()); /* 4 */
12
13
       Point[] ps = pc.getPointsInQuadrantI();
       System.out.println(ps.length); /* 1 */
14
       System.out.println("(" + ps[0].getX() + ", " + ps[0].getY() + ")"); /* (3, 4) */
15
16
     }
17
   }
```

Figure 3: Tester for Instances of ${\tt Point}$ and ${\tt PointTester}$

• Line 3 of PointTester

PointCollector pc = new PointCollector();

As explained in the constructor of **PointCollector** (Section 2, page 3), the above line results in the following object structure:



Therefore, executing the print statement System.out.println(pc.getNumberOfPoints()) in Line 4 of PointCollector outputs 0 to the console.

• Line 5 of PointTester

pc.addPoint(3, 4);

This method call has the context object pc, declared as an address holder of some PointCollector object, and the mutator method void addPoint(double x, double y) (defined in Lines 7 to 10 in Figure 2, Section 2, page 3) is instantiated (by substituting this by pc, x by 3 and y by 4) as:

```
public void addPoint(3, 4) {
    /* current value of nop is 0 */
    pc.points[pc.nop] = new Point(3, 4);
    pc.nop++;
    /* current value of nop is 1 */
}
```

Executing the above method call results in the following object structure:



Therefore, executing the print statement System.out.println(pc.getNumberOfPoints()) in Line 6 of PointCollector outputs 1 to the console.

• Line 7 of PointTester

pc.addPoint(-3, 4);

This method call has the context object pc, declared as an address holder of some PointCollector object, and the mutator method void addPoint(double x, double y) (defined in Lines 7 to 10 in Figure 2, Section 2, page 3) is instantiated (by substituting this by pc, x by -3 and y by 4) as:

```
public void addPoint(-3, 4) {
    /* current value of nop is 1 */
    pc.points[pc.nop] = new Point(-3, 4);
    pc.nop++;
    /* current value of nop is 2 */
}
```

Executing the above method call results in the following object structure:



Therefore, executing the print statement System.out.println(pc.getNumberOfPoints()) in Line 8 of PointCollector outputs 2 to the console.

• Line 9 of PointTester

pc.addPoint(-3, -4);

This method call has the context object pc, declared as an address holder of some PointCollector object, and the mutator method void addPoint(double x, double y) (defined in Lines 7 to 10 in Figure 2, Section 2, page 3) is instantiated (by substituting this by pc, x by -3 and y by -4) as:

```
public void addPoint(-3, -4) {
    /* current value of nop is 2 */
    pc.points[pc.nop] = new Point(-3, -4);
    pc.nop++;
    /* current value of nop is 3 */
}
```

Executing the above method call results in the following object structure:



Therefore, executing the print statement System.out.println(pc.getNumberOfPoints()) in Line 10 of PointCollector outputs 3 to the console.

• Line 11 of PointTester

pc.addPoint(3, -4);

This method call has the context object pc, declared as an address holder of some PointCollector object, and the mutator method void addPoint(double x, double y) (defined in Lines 7 to 10 in Figure 2, Section 2, page 3) is instantiated (by substituting this by pc, x by 3 and y by -4) as:

```
public void addPoint(3, -4) {
    /* current value of nop is 3 */
    pc.points[pc.nop] = new Point(3, -4);
    pc.nop++;
    /* current value of nop is 4 */
}
```

Executing the above method call results in the following object structure:



Therefore, executing the print statement System.out.println(pc.getNumberOfPoints()) in Line 12 of PointCollector outputs 4 to the console.

Pause and think: Up to this point, **nop** == **4** indicates that there have been four points collected in the **points** array of the point collector object **pc**. We observe that out of these four points collected, only the points stored at **pc.points**[**0**] is located in the first quadrant. Therefore, we would expect the return value from a subsequent (accessor) method call **pc.getPointsInQuadrantI()** to be an array of length **1**.

• Line 13 of PointTester

```
Point[] ps = pc.getPointsInQuadrantI();
```

This method call has the context object pc, declared as an address holder of some PointCollector object, and the accessor method Point[] getPointsInQuadrantI() (defined in Lines 11 to 27 in Figure 2, Section 2, page 3) is executed (by substituting this by pc, nop by its current value 4 and points by the array above where the first four slots are occupied) as:

```
Lines 12 to 20 of PointCollector
12
     Point[] ps = new Point[pc.nop]; /* current value of nop is 4 */
13
     int count = 0; /* number of points in Quadrant I */
14
     for(int i = 0; i < pc.nop; i ++) {
15
       Point p = points[i];
       if(p.getX() > 0 && p.getY() > 0) {
16
17
         ps[count] = p;
18
         count ++;
19
       }
20
     }
```

Right after Line 13, we initialize an array **ps** of size **nop** (whose current value is 4, meaning that at most the four points collected are all located in the first quadrant) and a counter **count** (whose job is to keep track of out of those points collected so far, how many of them are located in the first quadrant). Here is a visual summary of the object structure after right after executing Line 13:



Then, the for loop in Lines 14 to 20 is executed to iterate through pc.points[0], pc.points[1], pc.points[2], and pc.points[3], and store whichever that is located in the first quadrant into slots in ps. Only the first iteration (when i == 0) which inspects pc.points[0] would pass the condition in Line 16, store the address of the point object (3, 4) into ps, and update the count value accordingly; all later three iterations (when i = 1, 2, 3, 3), the condition in Line 6 evaluates to *false*, so none of those three points would be stored in ps and the value of count would not be incremented any further.

Here is a visual summary of the object structure after right after executing Line 20:



Remarks. It is important to notice that in Line 14, the value of **nop** (which is currently 4), rather than the value of pc.points.length (which has remained 100 since it was first created), is used as the upper bound of the loop counter. What if i < pc.points.length was used instead as the stay condition of the **for** loop? Unfortunately, when the value of loop counter **i** reaches **4** (which corresponds to the first, left-most index at which pc.points stores a null value), we would still pass this (wrong) stay condition i < pc.points.length and enter the body of the for loop. Consequently, executing Line 15 assigns null to variable p, and when the execution reaches Line 16, executing p.getX() and p.getY() is equivalent to executing null.getX() and null.getY(), which would trigger a NullPointerException.

Lines 21 to 25 of PointCollector

```
21
     Point[] q1Points = new Point[count];
22
     /* ps contains null if count < nop */
23
     for(int i = 0; i < count; i ++) {
24
       q1Points[i] = ps[i]
     }
```

25

From Lines 12 to 20, we have updated the values of **count** (which is now 1) and **ps**, so that array ps now has 1 non-null slot and 3 (calculated through ps.length - count, where ps.length == **nop && nop == 4**) null slots. We cannot simply return array **ps** as the return value for method getPointsInQuadrantI, as it contains null slots. As a result, we execute Lines 21 to 25 to initialize an array **q1Points** whose length corresponds to exactly those (indicated by the value of **count**) collected points that are located in the first quadrant, and we simply copy over the addresses of the first-quadrant points to slots in array **q1Points**.

Figure 4 summarizes the object structure after right after executing Line 25:



Figure 4: Final Object Structure from Executing PointCollectorTester

Again, it is important to notice that in Line 23, the value of **count** (which is currently 1), rather than the value of **nop** (which is 4), is used as the upper bound of the loop counter. What if i < inop was used instead as the stay condition of the for loop? Unfortunately, when the value of loop counter i reaches 1, we would still pass this (wrong) stay condition i < nop and enter the body of the for loop. Consequently, executing the expression q1Points[i] in Line 24 would trigger an ArrayIndexOutOfBoundException because the array q1Points was created in Line 21 to have length count (which is 1).

Line 26 of PointCollector

26 **return** q1Points;

Up to now, there are three candidate objects (whose types all match the return type Point[] of the accessor method getPointsInQuadrantI) that can be the return values:

1. pc.points

This should not be returned, since it contains pc.points.length - nop slots that store null values, and even for those slots that store addresses of Point objects, some of them are actually not located in the first quadrant.

2. ps

This should not be returned, since it contains **nop** - **count** slots that store **null** value.

3. q1Points

This should be returned, since it contains exactly those collected points that are located in the first quadrant, and there are no **null**-slots in this array.

Therefore, we return q1Points (which stores the address of the array of size 1) as the return value of accessor method getPointsInQuadrantI. This return value is stored, from Line 13 of PointCollectorTester (Figure 3, page 8), into variable ps. Consequently:

- * In Line 14 of PointCollectorTester, the print statement System.out.println(ps.length) outputs 1 to the console.
- * In Line 15 of PointCollectorTester, the print statement System.out.println("(" + ps[0].getX() + ", " + ps[0].getY() + ")") outputs (3, 4) to the console.
- Exercise. Figure 3 (page 8) above shows declarations and manipulations in the context of a console application. As an exercise, convert it into a JUnit test method. For example, Line 3 and 4 may be converted to:

```
@Test
```

```
public void test() {
    PointCollector pc = new PointCollector();
    assertEquals(0, pc.getNumberOfPoints());
}
```