Programming Exercise: Implementing Selection Sort

For the following assignments, you will start with the files provided, using most of the classes, and modifying only one of them.

First there are several classes provided from the previous lesson that are unchanged:

- The class **Location**, from the Android platform and revised for this course, a data class representing a geographic location. One of the constructors has parameters latitude and longitude, and one of the public methods is distanceTo.
- The class **QuakeEntry**, from the lesson, which has a constructor that requires latitude, longitude, magnitude, title, and depth. It has several get methods and a **toString** method.
- The class **EarthQuakeParser**, from the lesson, which has a **read** method with one String parameter that represents an XML earthquake data file and returns an ArrayList of QuakeEntry objects.

There are several new classes:

- The **QuakeSortWithTwoArrayLists** class includes code shown in the video “Translating to Code” on how to sort using two ArrayLists of QuakeEntry’s. You should run the **testSort** method and understand how this sorting algorithm works.
- The **QuakeSortInPlace** class is the class you will modify. Currently it implements the selection sort algorithm that sorts earthquakes by magnitude in place, in one ArrayList of QuakeEntry’s. The code for the method **sortByMagnitude** was shown in the video “In Place.” You should run the **testSort** method and understand how this sorting algorithm works.

Assignment 1: Sort by Depth

In this assignment, you will add methods in the **QuakeSortInPlace** class to sort the quakes by depth, from largest depth to smallest depth. This will mean any positive depth values will be first, followed by depths with increasingly negative values, e.g., 200.00, 0.00, -20000.00, -100000.00.

Specifically, for this assignment, you will modify the **QuakeSortInPlace** class:

- Write the method **getLargestDepth** that has two parameters, an ArrayList of type QuakeEntry named **quakeData** and an int named **from** representing an index position in the ArrayList. This method returns an integer representing the index position of the QuakeEntry with the largest depth considering only those QuakeEntry's from position **from** to the end of the ArrayList.
- Write the void method **sortByLargestDepth** that has one parameter, an ArrayList of type QuakeEntry named **in.** This method sorts the QuakeEntry's in the ArrayList by depth using the selection sort algorithm, but in reverse order from largest depth to smallest depth (the QuakeEntry with the largest depth should be in the 0th position in the ArrayList). This method should call the method **getLargestDepth** repeatedly until the ArrayList is sorted.
- Modify the **testSort** method to comment out the line **sortByMagnitude** and add below this line a call to **sortByLargestDepth**. Run your program on any data file and you should see the earthquakes in sorted order from largest depth to smallest depth.

Assignment 2: Bubble Sort

For this assignment, you will implement the bubble sort algorithm, which also sorts the data in an ArrayList in place. Bubble sort sorts using several passes. One pass over the ArrayList compares adjacent elements and swaps them if they are out of order. For example, if we want to sort the numbers:

```
  4 7 3 8 1
```

One pass would compare the first two elements in the ArrayList, 4 and 7. They are in order, so there is no swap. Then the second and third elements are compared, 7 and 3, they are out of order, so they would be swapped, resulting in:

```
  4 3 7 8 1
```

The next two adjacent elements are compared, the third and fourth elements, 7 and 8. They are in order, so there is no swap. The next two adjacent elements are 8 and 1. They are out of order, so they are swapped, resulting in:

```
 4 3 7 1 8
```

The pass is complete. Notice that as a result of the pass, the largest element in the ArrayList, which is larger than any other element, is “bubbled” down and is the last item in the ArrayList, where it belongs in the final sorted order.

After a second pass, the elements in the ArrayList should be in the order:

```
  3 4 1 7 8
```

Notice now that the last two elements, the largest two, should be where they belong in the final sorted order.

The bubble sort algorithm works as follows. If there are N elements in the ArrayList. Apply N-1 passes, and the ArrayList should be sorted. Notice that with each pass, there is one more element where it belongs in the final sorted order. Thus each pass can look at one fewer element than the previous pass.

Specifically, for this assignment, you will modify the **QuakeSortInPlace** class:

- Write the void method **onePassBubbleSort** that has two parameters, an ArrayList of type QuakeEntry named **quakeData** and an int named **numSorted** that represents the number of times this method has already been called on this ArrayList and thus also represents the number of the elements that are guaranteed to already be where they belong when the ArrayList is sorted by magnitude. This method makes one pass of bubble sort on the ArrayList. It should take advantage of the fact that the last **numSorted** elements are already in sorted order.

- Write the void method **sortByMagnitudeWithBubbleSort** that has one parameter, an ArrayList of type QuakeEntry named **in.** If the ArrayList in has N elements in it, this method should call **onePassBubbleSort** N – 1 times to sort the elements in **in**.
- Modify the **testSort** method to comment out the line **sortByLargestDepth**, and add below this line a call to **sortByMagnitudeWithBubbleSort**. Run your program on any data file and you should see the earthquakes in sorted order from smallest to largest magnitude.
- Are you convinced your program is working correctly? Let's add more output and test it on a small file. Add code to **sortByMagnitudeWithBubbleSort** to print all the quakes before a pass, and then to print all the quakes after each pass, identifying the pass. Since there will be a lot of data, you'll only want to run your program on a small file. Once you are sure it works, you probably want to then comment out the print statements. Run your program on the file **earthquakeDataSampleSix2.atom**, which has data on only five earthquakes. You should get the following output. Focusing on the magnitude, after pass 0 (the first pass) the quake with magnitude 4.80 is last, after pass 1 the quake with magnitude 2.60 is where it belongs. Note that pass 3 wasn't needed as the quakes were already in sorted order. That might happen sometimes.

Output:

```
read data for 5 quakes
(-23.27, -67.66), mag = 4.80, depth = -175320.00, title = 69km SE of San Pedro de
Atacama, Chile
(35.68, -118.10), mag = 1.50, depth = -8280.00, title = 27km W of Inyokern, California
(36.22, -117.89), mag = 2.60, depth = -1450.00, title = 12km ESE of Olancha,
California
(36.95, -121.57), mag = 1.00, depth = -8660.00, title = 6km S of Gilroy, California
(38.82, -122.77), mag = 1.40, depth = -1300.00, title = 3km W of Cobb, California
Printing Quakes after pass 0
(35.68, -118.10), mag = 1.50, depth = -8280.00, title = 27km W of Inyokern, California
(36.22, -117.89), mag = 2.60, depth = -1450.00, title = 12km ESE of Olancha,
California
(36.95, -121.57), mag = 1.00, depth = -8660.00, title = 6km S of Gilroy, California
(38.82, -122.77), mag = 1.40, depth = -1300.00, title = 3km W of Cobb, California
(-23.27, -67.66), mag = 4.80, depth = -175320.00, title = 69km SE of San Pedro de
Atacama, Chile
Printing Quakes after pass 1
(35.68, -118.10), mag = 1.50, depth = -8280.00, title = 27km W of Inyokern, California
(36.95, -121.57), mag = 1.00, depth = -8660.00, title = 6km S of Gilroy, California
(38.82, -122.77), mag = 1.40, depth = -1300.00, title = 3km W of Cobb, California
(36.22, -117.89), mag = 2.60, depth = -1450.00, title = 12km ESE of Olancha,
California
```

```
(-23.27, -67.66), mag = 4.80, depth = -175320.00, title = 69km SE of San Pedro de
Atacama, Chile
Printing Quakes after pass 2
(36.95, -121.57), mag = 1.00, depth = -8660.00, title = 6km S of Gilroy, California
(38.82, -122.77), mag = 1.40, depth = -1300.00, title = 3km W of Cobb, California
(35.68, -118.10), mag = 1.50, depth = -8280.00, title = 27km W of Inyokern, California
(36.22, -117.89), mag = 2.60, depth = -1450.00, title = 12km ESE of Olancha,
California
(-23.27, -67.66), mag = 4.80, depth = -175320.00, title = 69km SE of San Pedro de
Atacama, Chile
Printing Quakes after pass 3
(36.95, -121.57), mag = 1.00, depth = -8660.00, title = 6km S of Gilroy, California
(38.82, -122.77), mag = 1.40, depth = -1300.00, title = 3km W of Cobb, California
(35.68, -118.10), mag = 1.50, depth = -8280.00, title = 27km W of Inyokern, California
(36.22, -117.89), mag = 2.60, depth = -1450.00, title = 12km ESE of Olancha,
California
(-23.27, -67.66), mag = 4.80, depth = -175320.00, title = 69km SE of San Pedro de
Atacama, Chile
EarthQuakes in sorted order:
(36.95, -121.57), mag = 1.00, depth = -8660.00, title = 6km S of Gilroy, California
(38.82, -122.77), mag = 1.40, depth = -1300.00, title = 3km W of Cobb, California
(35.68, -118.10), mag = 1.50, depth = -8280.00, title = 27km W of Inyokern, California
(36.22, -117.89), mag = 2.60, depth = -1450.00, title = 12km ESE of Olancha,
California
(-23.27, -67.66), mag = 4.80, depth = -175320.00, title = 69km SE of San Pedro de
Atacama, Chile
```

Assignment 3: Check for Completion

For this assignment, we will explore the idea that the data might be sorted before all the passes are complete. We want to write a method to determine whether a list is already sorted and then use that method to end our sorting algorithms early if earthquakes are sorted early.
Specifically, for this assignment, you will modify the **QuakeSortInPlace** class:

- Write the method **checkInSortedOrder** that has one parameter, an ArrayList of type QuakeEntry named **quakes.** This method returns true if the earthquakes are in sorted order by magnitude from smallest to largest. Otherwise this methods returns false. You'll need to loop through the ArrayList and check adjacent earthquakes to see if any are out of order.
- Write the void method **sortByMagnitudeWithBubbleSortWithCheck** that has one parameter, an ArrayList of type QuakeEntry named **in.** If the ArrayList in has N elements in it, this method should call **onePassBubbleSort** at most N – 1 times. This method should call **checkInSortedOrder** and stop early if the ArrayList is already sorted. This method should print how many passes were needed to sort the elements.
- Modify the **testSort** method to call to **sortByMagnitudeWithBubbleSortWithCheck**. Run your program on any data files **earthquakeDataSampleSix1.atom** (should sort after 2 passes) and **earthquakeDataSampleSix2.atom** (should sort after 3 passes). Both of these files have five earthquakes.
- Write the void method **sortByMagnitudeWithCheck** that has one parameter, an ArrayList of type QuakeEntry named **in.** This method sorts earthquakes by their magnitude from smallest to largest using selection sort similar to the **sortByMagnitude** method. However, this method should call **checkInSortedOrder** and stop early if the ArrayList is already sorted. This method should print how many passes were needed to sort the elements. For selection sort, one pass has exactly one swap.
- Modify the **testSort** method to call to **sortByMagnitudeWithCheck**. Run your program on any data files **earthquakeDataSampleSix1.atom** (should sort after 3 passes) and **earthquakeDataSampleSix2.atom** (should sort after 4 passes). Both of these files have five earthquakes.