COS 126 **General Computer Science** **Fall 2017**

Programming Exam 2

Instructions. This exam has only one part. You have 50 minutes. The exam is *open course materials*, which includes the course textbook, the companion booksite, the course website, your course notes, and code you wrote for the course. Accessing other information or communicating with a non-staff member (such as via email, text, Facebook, Piazza, phone, or Snapchat) is prohibited.

Setup. You may download the necessary files from the *Class Meetings* page now.

Submission. Submit your solution via the link on the *Class Meetings* page. Click the *Check All Submitted Files* button to verify your submission. You may submit multiple times.

Grading. Your program will be graded for correctness, clarity (including comments), design, and efficiency. You will receive partial credit for a program that correctly implements some of the required functionality. You will receive a substantial penalty if your program does not compile or if you do not follow the prescribed input/output specifications.

Discussing this exam. Discussing or communicating the contents of this exam before solutions have been posted is a violation of the Honor Code.

This exam. You must turn in this physical exam. Print your name, NetID, precept, and the room in which you are taking the exam in the space below. Also, write and sign the Honor Code pledge. You may fill in this information now.

Name:

NetID:

Exam Room:

Precept:

"I pledge my honor that I will not violate the Honor Code during this examination."

Signature

Problem: Compute Geographical Path Statistics. (30 Points) Complete the given `Path.java` file so that it meets the specifications below. The `main` method is done for you and must not be modified. Use the given `Coordinate.java` and `Stats.java` without modification.

Input Specification. The input from standard input consists of zero or more lines with three doubles per line separated by a space. Each line is a geographical Earth coordinate: a *longitude in decimal degrees*, a *latitude in decimal degrees*, and an *altitude in meters*. These coordinates together represent a path from the first coordinate to the last coordinate.

```
% more COS126Walk.txt
-74.657080  40.348197 66.0  <-- McCosh 10
-74.652250  40.350259 57.0  <-- CS Building
Longitude   Latitude  Altitude
```

The file `COS126Walk.txt` contains two coordinates: the location of McCosh 10, the start point of the path, and the location of the CS Building, the end point of the path.

For each line in the input, the given `main` method reads the coordinates, creates a `Coordinate` object (defined in `Coordinate.java`), and calls your `addWaypoint` method to add the coordinate to the path object (created initially by a call to the constructor). Your `addWaypoint` method must add the `Coordinate` objects to a linked list of your own design.

As described below, program behavior changes depending on the presence of an *optional* integer command line argument. Your program need not handle non-conforming inputs.

Output Specification *Without* Command Line Argument. (20 of the 30 points) If no command line argument is provided, `main` prints the path using a call to the `Path.toString` method that you must implement. For each coordinate in the path, your `Path.toString` method should include the result of calling `Coordinate.toString` on that coordinate followed by a newline. (If the path is empty, print nothing.) Example output:

```
% java-introcs Path < COS126Walk.txt
-74.65708,40.348197,66.0
-74.65225,40.350259,57.0
```

Output Specification *With* Command Line Argument. (10 of the 30 points) If invoked with an integer command line argument (an altitude threshold described below), `main` prints path statistics computed by calling your `Path.java` `computeStats` method. The output for `COS126Walk.txt` is:

```
                          Altitude Threshold
% java-introcs Path 1 < COS126Walk.txt
Points:    2
Distance:  469 meters
High:      66 meters
Low:       57 meters
Tot. Asc:  0 meters
Tot. Desc: 9 meters
```

Your `computeStats` method must calculate these statistics, store them in a `Stats` object (created by calling the provided `Stats.java` class constructor with these values), and return

a reference to this newly created `Stats` object. The provided `Path.java` contains a dummy `computeStats` that returns zeros for all statistics. The statistics to compute are:

`Points:` Number of geographical coordinate points in the path.

`Distance:` The distance of the path. Use the provided `Coordinate.haversineDist` method to compute the distance between two coordinates using the Haversine formula.[1]

`High:` The maximum of all coordinate altitudes.

`Low:` The minimum of all coordinate altitudes.

`Tot. Asc:` The total ascent is the sum of all altitude increases. Total ascent ignores all altitude decreases. Total ascent is subject to the altitude threshold as described below.

`Tot. Desc:` The total descent is the sum of all altitude decreases. Total descent ignores all altitude increases. Total descent is subject to the altitude threshold as described below.

Calculating Total Ascent/Descent with Altitude Threshold. Consider the following altitude sequence:

```
Measured:  10, 11,  9, 11, 10, 11, 12, 11, 14, 15, 17, 14
Asc/Desc:  --, +1, -2, +2, -1, +1, +1, -1, +3, +1, +2, -3
```

Assuming that these measured altitudes are accurate, the total ascent is `1 + 2 + 1 + 1 + 3 + 1 + 2 = 11` and the total descent is `2 + 1 + 1 + 3 = 7`.

Unfortunately, GPS altitude measurements are notoriously inaccurate, and you are being provided with input files with coordinates collected by GPS. Unaddressed, erroneous fluctuations in altitude will be counted as actual climbing and descending, artificially inflating total ascent/descent. To address this issue, GPS path calculators will only *note* vertical movements when the magnitude of such movement exceeds a threshold value. You will do the same with the altitude threshold given on the command line in your `Path.java` `calculateStats` method.

To illustrate threshold usage, consider the same sequence with an altitude threshold of 2:

```
Measured:  10, 11,  9, 11, 10, 11, 12, 11, 14, 15, 17, 14
Noted:     10, --, --, --, --, --, --, --, 14, --, 17, 14
Asc/Desc:  --, --, --, --, --, --, --, --, +4, --, +3, -3
```

Using only the noted measurements, the total ascent becomes `4 + 3 = 7` and total descent becomes `3`. This is likely closer to the truth.

The altitude sequence used in this example is given in the provided `Levitate.txt`. You can use this file to test your program with altitude thresholds of 0 and 2 as in the above example. (There is no horizontal movement recorded among `Levitate.txt`'s 12 points, so path distance is 0 meters.)

Note: the altitude threshold is not used to calculate the altitude high and low statistics.

[1] Given two points, the Haversine formula computes the great-circle distance on a sphere. The Earth is not a sphere, the Haversine formula doesn't compensate for increased horizontal distance at altitude, and it is not ideal for short distances, but it is good enough for our purposes.

Requirements.

- You must store all coordinates as `Coordinate` class objects in a linked list of your own design.
- You must implement `Path.addWaypoint`, `Path.computeStats`, and `Path.toString`. You may modify the `Path` constructor and add private variables and classes.
- Your program must take no longer than a few seconds to run on any of the given inputs.

Suggestion. Do the linked list implementation, `addWaypoint`, and `toString` first. These are worth the most points. Do the ascend/descend with threshold computation last after everything else is tested and working. The points return on effort for that part is the lowest.

Other Inputs for Testing. Assume that the submission scripts will not perform any significant testing for you. You may use the following inputs to test your code on your computer. You may also want to test other conforming inputs such as a zero coordinate path (all statistics are zero) and a one coordinate path (number of points is one, high and low are equal to coordinate altitude, and all other statistics are zero).

`NickMarathon.txt` - the NYC Marathon as run by Professor Nick Feamster. Threshold is set to 7 meters.

```
% java-introcs Path 7 < NickMarathon.txt
Points:    2034
Distance:  42523 meters
High:      57 meters
Low:       -1 meters
Tot. Asc:  125 meters
Tot. Desc: 136 meters
```

`DavidHike.txt` - a hike over Sugarloaf Mountain in the Catskills by Professor David August. Threshold is set to 15 meters.

```
% java-introcs Path 15 < DavidHike.txt
Points:    2311
Distance:  12836 meters
High:      1165 meters
Low:       461 meters
Tot. Asc:  639 meters
Tot. Desc: 752 meters
```

Submission. Submit only `Path.java` via the link on the *Class Meetings* page.

After the Exam. If you would like to view or examine paths using a tool like Google Earth, you can produce KML files by printing the strings produced by calling `Coordinate.KMLHeader`, `Path.toString`, and `Coordinate.KMLFooter` in that order.