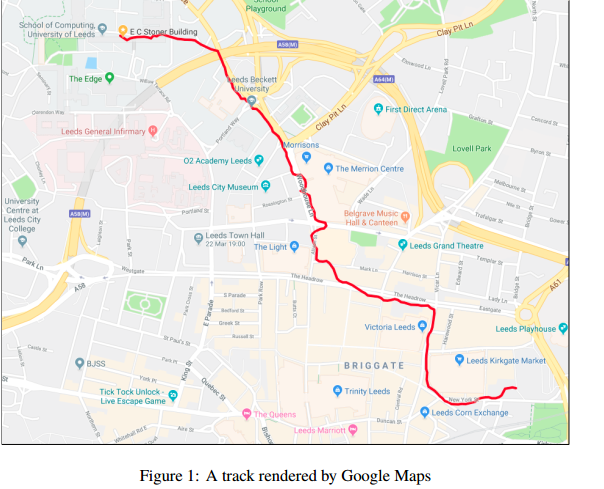
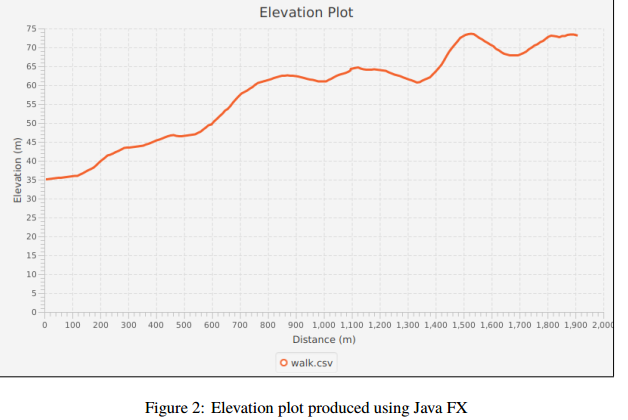
COMP1721 Object-Oriented Programming  
Coursework 1: Creating & Using Classes  
1 Introduction  
This assignment assesses your ability to implement classes and use them in a small program.  
Consider the GPS data generated by a device such as a mobile phone. Your current location is represented  
as a point, consisting of a timestamp, a longitude (in degrees), a latitude (in degrees) and an elevation above sea level (in metres). Movement while GPS is enabled generates a track: a sequence of points representing successive samples from the GPS sensor.  
Your task is to implement classes that can be used to represent points and tracks, along with a small program that demonstrates use of the classes. To get you started, we have provided you with a partial implementation of one of these classes, named Point, along with another class to represent exceptions that can occur in your code.  
2 Preparation  
 It is important that you follow the instructions below precisely. We strongly recommend that you perform these steps in a Linux environment.

**1**. Download cwkfiles1.zip from Minerva and put it into the top level of your repository—i.e., the directory containing the cwk1, cwk2 and exercises subdirectories.  
**2**. Open a terminal window at the top-level of your repository, then unpack the Zip archive using the command unzip cwkfiles1.zip. This should create subdirectories immediately below cwk1 named data, lib, src and test. Make sure that this is exactly what you see! For example, you should not have a subdirectory of cwk1 that is itself named cwk1.  
All of your code should be written in files in the src subdirectory. The other subdirectories should be left untouched. Be aware that you may lose marks if you alter anything in data, lib or test, or if you have the wrong directory structure to start with.3 Basic Solution  
This is worth 18 marks.  
Please read all three of the subsections below before starting work. We also recommend that you gain some experience of implementing classes by doing Exercise Set 3 before you start.  
3.1 Point Class  
Complete the implementation of the Point class that we have provided by adding  
 • A field to represent the timestamp, of type ZonedDateTime (see below)  
 • Fields to represent longitude, latitude and elevation, all of type double  
 • A constructor that initialises all fields, using the values supplied as parameters (see below)  
1  
 • Getter methods for the fields, named getTime, getLongitude, getLatitude, getElevation (you must use these exact names)  
 • A toString method that returns a string representation of a Point looking like this:  
 (-1.54853, 53.80462), 72.5 m  
 (The values here are longitude, then latitude, then elevation.)  
ZonedDateTime is a class from Java’s standard Date/Time API, in the package java.time—see the API documentation for further details.  
Make sure that it is not possible to create a Point object with an invalid latitude or longitude. Use the constants provided in the class to help you with this, and throw an instance of the provided exception class, GPSException, if inappropriate coordinates are supplied (see the Time class discussed in lectures for an example of this).  
3.2 Track Class  
In a new file called Track.java, Write a class called Track that can store a sequence of Point objects. Give this class  
 • A field suitable for storing a sequence of Point objects  
 • A default constructor that creates a track containing no points  
 • A readFile method that creates a sequence of Point objects from data in a file, the name of which is supplied as a String parameter (see below)  
 • A method named add that adds a new point to the end of the track  
 • A method named size that returns the number of points currently stored in the track  
 • A method named get that returns the Point object stored at a given position in the sequence, specified as an int parameter (see below)  
The readFile method will need to read CSV files, examples of which can be found in the data directory. It should use a Scanner to do this. A good approach here would be to read the file line-by-line, split up the line on commas, then parse each item separately. You can use the static method parse of the ZonedDateTime class to parse the timestamp.  
readFile should not catch any exceptions that might occur during reading of the file. It will need an exception specification, declaring that it could throw FileNotFoundException if the named file cannot be accessed. Your implementation should throw a GPSException if any record within the file doesn’t contain the exact number of values needed to create a Point object.  
The get method should use the int value passed to it as an index into the sequence of Point objects, but before that the method should check this int value and throw an instance of GPSException if it is not within the allowed range.  
3.3 Testing Your Solution  
To help you check whether you’ve implemented these classes as required, we have provided some code that runs a series of tests on the classes. To run the tests from a terminal window, cd into the test directory and enter the following:  
 **./basictests**  
This will work on Linux and in a macOS terminal. It should work in Windows Subsystem for Linux and may work in a Git Bash shell, although this hasn’t been tested. Running the tests from within IntelliJ should also be possible; advice on how to do this will be provided separately.  
We strongly recommend you run these tests before implementing the rest of your solution and before submitting anything. Your mark will be based partly on the number of these tests that pass.  
Note also that these tests won’t compile until all of the methods listed above exist in their respective classes.  
So if you want to run the tests successfully, you will need to provide stubs—dummy implementations—for any methods that you haven’t implemented yet. The stub for a void method can have an empty method body. The stub for a method that returns a value can just return a constant of the appropriate type (e.g., 0 in  
the case of numeric return types).  
4 Full Solution  
This is worth a further 12 marks.  
4.1 Track Class  
Add the following to the Track class:  
 • Methods lowestPoint and highestPoint that return the Point objects having the lowest and highest  
elevations, respectively  
 • A method totalDistance that returns the total distance travelled in metres when moving from point  
to point along the entire length of the track  
 • A method averageSpeed that returns the average speed along the track, in metres per second  
All of these methods should throw a GPSException if aren’t enough points to do the necessary computation.  
To implement totalDistance, you will need to compute ‘great-circle distance’ between adjacent points on the track. A method to do this already exists in the Point class. Given two Point objects, p and q, the great-circle distance in metres between them (ignoring elevation) will be given by  
 double distance = Point.greatCircleDistance(p, q);  
To implement averageSpeed you will need to compute the amount of time that has passed between measurements for the first and last points on the track. You can use the ChronoUnit type for this—specifically, the between method, which can be called on ChronoUnit.SECONDS to measure time intervals in units of seconds.  
4.2 Testing  
The same considerations apply here as for the Basic Solution. You can run the tests with  
 ./fulltests  
4.3 TrackInfo Program  
In a file named TrackInfo.java create a small program that creates a Track object from data in a file whose name is provided as a command line argument. You program should display: the number of points in the track; its lowest and highest points; the total distance travelled; and the average speed.  
When your program is run on the file walk.csv, provided in the data directory, it should generate output very similar to this:  
 194 points in track  
 Lowest point is (-1.53637, 53.79680), 35.0 m  
 Highest point is (-1.54835, 53.80438), 73.6 m  
 Total distance = 1.904 km  
 Average speed = 1.441 m/s  
If no filename is supplied on the command line, your program should print a helpful error message and then use System.exit to terminate, with a non-zero value for exit status.  
The program should intercept any exceptions that occur when reading from the file or performing computation. The program should print the error message associated with the exception and then use System.exit to terminate, with a non-zero value for exit status.  
  
5 Advanced Tasks  
For a few extra marks, implement ONE of two options suggested below.  
These tasks are more challenging and will require additional reading/research; attempt them only if you manage to complete the previous work fairly quickly and easily.  
5.1 Option 1: KML Files  
This is worth an additional 2 marks.  
1. Add to the Track class a new method named writeKML. This should have a single String parameter,representing a filename. It should write track data to the given file, using Google’s Keyhole Markup Language format.  
2. Modify TrackInfo so that it calls writeKML to generate a KML file.  
3. Generate a KML file for the track represented by walk.csv. Visualise the file by uploading it to Google Maps (see Figure 1) or by importing it into Google Earth. Grab a screenshot of the result and place it in the cwk1 directory so that it will be included in your submission.

  
  
5.2 Option 2: Elevation Plot  
This is worth an additional 4 marks.  
 1. Investigate JavaFX by reading Chapter 14 of the Liang book and trying out some of its examples, or by studying the documentation at  
 https://docs.oracle.com/javase/8/javase-clienttechnologies.htm  
Pay particular attention to the section in the documentation on drawing charts.  
 2. Write a program called ElevationPlot that uses Java FX to plot elevation as a function of distance along a track. Figure 2 shows an example of what this could look like.  
  
  
  
6 Marking  
40 marks are available for this assignment.  
A basic solution can earn up to 24 marks (60% of those available); a full solution can earn up to 36 marks  
(90% of those available).  
Mark allocation breaks down as follows:  
18 Tests for basic solution  
8 Tests for full solution  
4 TrackInfo program  
4 Advanced tasks  
4 Coding style and comments  
2 Use of version control