Jim is desperate for coffee. However, Jim likes a special coffee that is a combination of ingredients, and these ingredients are located at various coffee shops around Ames. In addition, in order to get that coffee just right, the ingredients must be picked up in a certain order so they can be mixed with previous ingredients immediately for that perfect flavor. Jim needs your help to collect these ingredients and deliver them to him as quickly as possible and procured in the right order so he can make the perfect cup of coffee. Also, Jim has this crazy idea of providing pipes all over Ames so that every building can have coffee on tap. Jim also wants to create a coffee delivery business that delivers perfect coffee to your home or office, just in case the city doesn't approve his coffee pipe dream. (punn intended)

Project Description

This is a programming project that must be written in Java and turned in via Blackboard. The project has several parts, including bonus questions to help you makeup points you might have lost on previous homework assignments. This assignment utilizes the Graph and GraphAlgorithms Java classes that you created in Assignment 7.

In the first and second parts, you will write code to find quick routes through the city of Ames to help get Jim his special coffee as soon as possible.

The third part of the assignment you will write code to help Jim estimate the cost of adding coffee pipes all over the city of Ames.

Finally, for fourth and bonus part, you will help Jim determine estimates of quick delivery routes for his coffee delivery fleet of vehicles.

Part 1: (150 points)

For this part, copy all the files from HW 7 and place them in the packages cs311.hw8.graph and cs311.hw8.graphalgorithms appropriately. You will need to modify the package header in the files. To your GraphAlgorithms class, create a method called `ShortestPath` that determines the shortest path between to vertices in the graph. The signature for this method is

```
public static <V, E extends IWeight> List<Edge<E>> ShortestPath(
            IGraph<V, E> g, String vertexStart, String vertexEnd)
```

It is recommended that you throughly test this method on small graphs and then medium size graphs before moving on to the next sections. Be sure

to test edge cases, and other graphs with unique features. In the next sections you will be running your algorithms on graphs with more than 34,000 edges. As usual, you may post test graphs and JUnit tests to Blackboard for everyone to use. Some test cases we use to grade will come from what you post.

Documentation will be 15% of your grade.

Part 2: (150 points)

You are provided with a file named `AmesMap.txt` which encodes the city of Ames for use in part 2 of the project. This is an XML file downloaded directly from the Open Street Map website: `http://openstreetmap.org`. This file is a text XML with lots of data encoded. However, you are interested in two types of XML nodes. Nodes that are named `"node"` will represent places or points on a map and have among other attributes, a latitude and longitude. This XML nodes will become vertices in your graph with vertex data of the latitude and longitude of location the vertex represents. The other type of node you are interested in are called `"way"` nodes. These nodes will represent streets and correspond to edges in the directed graph. They will have edge data of distance between vertices, and the name of the street. It is **required** that you name your vertices in your graph the same as the "node id" in the XML data. If you do not do this we will not be able to grade this part or subsequent parts of the project.

Here is an example of the XML data from the actual Ames data file.

```
<node␣id="636696106"␣lat="42.0487220"␣lon="-93.5520650"␣version="1"
timestamp="2010-02-12T05:43:36Z"␣changeset="3853771"␣uid="232171"
user="Jeff␣Ollie's␣Bot"/>
<node␣id="636696108"␣lat="42.0486740"␣lon="-93.5520460"␣version="1"
timestamp="2010-02-12T05:43:36Z"␣changeset="3853771"␣uid="232171"
user="Jeff␣Ollie's␣Bot"/>
<node␣id="636696110"␣lat="42.0486270"␣lon="-93.5520180"␣version="1"
timestamp="2010-02-12T05:43:36Z"␣changeset="3853771"␣uid="232171"
user="Jeff␣Ollie's␣Bot"/>

<way id="16038319" version="4" timestamp="2012-10-01T13:28:53Z"
changeset="13322365" uid="590362" user="Mitchell Thomas">
<nd ref="161163268"/>
<nd ref="161169113"/>
<nd ref="161130647"/>
```

```
<nd ref="161169117"/>
<nd ref="161035496"/>
<nd ref="161169120"/>
<nd ref="161169121"/>
<nd ref="161169124"/>
<nd ref="161169127"/>
<tag k="highway" v="residential"/>
<tag k="name" v="South Wilmoth Avenue"/>
<tag k="tiger:cfcc" v="A41"/>
<tag k="tiger:county" v="Story, IA"/>
<tag k="tiger:name_base" v="Wilmoth"/>
<tag k="tiger:name_direction_prefix" v="S"/>
<tag k="tiger:name_type" v="Ave"/>
<tag k="tiger:reviewed" v="no"/>
<tag k="tiger:separated" v="no"/>
<tag k="tiger:source" v="tiger_import_dch_v0.6_20070810"/>
<tag k="tiger:tlid" v="65945844:65961902:65961903:65945810"/>
<tag k="tiger:upload_uuid" v="bulk_upload.pl-e80c81a2-0980-4acd-8909-7b56392d4de8"/>
<tag k="tiger:zip_left" v="50014"/>
<tag k="tiger:zip_right" v="50014"/>
</way>
```

The above shows 3 examples of an XML node with node name "node" and 1 example of an XML node with node name "way". In your project, all nodes named "node" are added to the graph as vertices with name given by the "id" attribute and data of "lat" and "lon". The "way" nodes will be used to populate the edges of your graph, however, only "nodes" with attributes of "highway" and "name" both present are added as an edge. If the node contains the attribute k = "oneway" and a value attribute v = "yes" then only an edge in one direction is added, otherwise two edges are added between the two vertices in both directions. The edge data stored with the edge is the distance in miles between the two locations the edge connects via two vertices and the street name given as the "v" (value) of the "name" attribute. Note that you will have to compute the distance by computing the distance between two latitude and longitude points on the earth. You may copy code from the Internet to do this provided you include a comment giving credit where you got the code or algorithm from, and the code is clearly delineated in your program.

Determining the two vertices in the edge comes from the "nd" node type (name). The "ref" attribute specifies a vertex id. In the example above,

there is an edge from vertex 161163268 and vertex 161169113. The edges continue to be specified by the chain: 161169113 and 161130647 all the way to the last edge between 161169124 and 161169127. In the above example, 8 edges are defined, all with street name data of "South Wilmoth Avenue" and distance computed by the latitude and longitude of the vertices.

You may write a parser to retrieve the data from the XML file, but using a library may be much easier. Here are some hints if you go the library route.

These are two very useful packages:

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
```

The following code can be used to create a "document" with elements that are XML nodes.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();

Document doc = builder.parse(f[0]);  // f[0] is a File type
doc.normalize();
```

The Java documentation at Oracle contains extensive documentation and examples on how to use these libraries. Their use will also be briefly discussed in lecture.

For this part of the assignment, you are to create a class named "OSMMap" that represents map data using the graph described above. Use appropriate inner classes to store the Vertex and Edge generic data. The class contains a default constructor that creates an empty map. It has a method named LoadMap(String filename) that loads an XML OSM file into a graph as described above. If map data is already present from a previous load, that data is cleared and the new map is loaded. Also add a method named "TotalDistance" that outputs the approximate miles of roadway in the map. Since there are very few oneway roads compared to normal roads we will approximate this by dividing the total edge distance in the graph by two. (This would not work well in a city like New York.) Finally, write a main method that outputs the approximate miles of road in Ames using the methods above and the data file AmesMap.txt.

Part 3: (150 points)

For this section include an inner class in the OSMMap class that defines a location as follows.

```
public static class Location
{
public double getLatitude()  // returns the latitude of the location
public double getLongitude() // returns the longitude of the location
}
```

In this section we will continue to add to the OSMMap class. Write a method named "ClosestRoad" that takes as input the latitude and longitude (as Location types) and returns the vertex ID that is closest to the specified latitude and longitude.

Next, write a method named "ShortestRoute" that takes as parameters two locations given by two parameters of Location type, and returns a list of String types that is the sequence of vertex ID names that gives the path. The signature for this method is:

```
public List<String> ShortestRoute(Location fromLocation,
                                  Location toLocation)
```

Note that the input parameters may not be a location of a vertex in the map data in which case the method should choose the closest vertex to the specified location to begin or end the route.

Now write a method named "StreetRoute" that takes as input a List¡String¿ of vertex ID names and returns a List¡String¿ of street names from the beginning location to the end location. Do not repeat street names that are consecutively the same in the list.

Finally, write a main method that helps Jim get that perfect cup of coffee. Write a main method that reads latitude and longitude values from a text file and outputs the sequence of street names (no consecutive duplicates) that gives the shortest route that travels through the latitude and longitude pairs in the order read from the file. The input file name and map file name is input by the user using a command line argument. For example, "java OSMMap mapfile.txt route.txt". The format of the route.txt file with n locations is

```
latitude_1 longitude_1
latitude_2 longitude_2
.
.
.
latitude_n longitude_n
```

Note that a simple Scanner can be used to read this file.

Part 4: Bonus (150 points)

Given a route.txt file as defined in Part 3, write a method named ApproximateTSP in the OSMMap class that takes as input a list of Strings that contains vertex IDs and outputs an approximate tour. A tour is a sequence of vertices where the first node is the start node, and every other node is visited exactly once and ends back at the start node. (This is the famous TSP problem.) Tthe trick is to find a minimum tour which is great for coffee deliveries. Your ApproximateTSP method must return a tour that is no more than double the distance of the optimal or best tour. Note that this may require a bit of research and reading to find an algorithm that will work. A good start might be to look at this page: `http://www.math.uwaterloo.ca/tsp/`.

As part of this section you may receive 20 bonus points (out of the 150) if you write a class Named "PipeDream" that correctly outputs the weightsum of the MST of the city of AMes graph using your graph minimum spanning tree method.