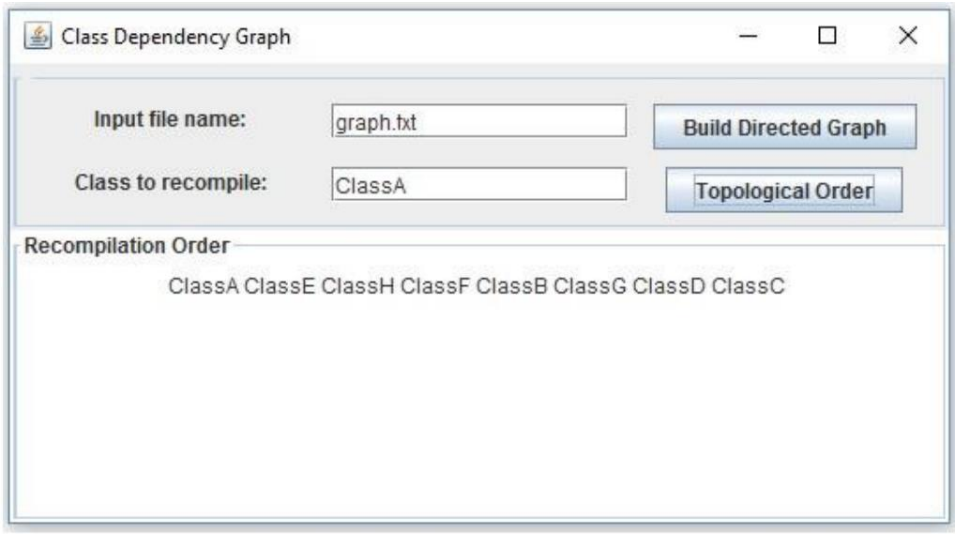
CMSC 350 Project 4

The fourth programming project involves designing, writing and testing a program that behaves like the Java command line compiler. Whenever we request that the Java compiler recompile a particular class, it not only recompiles that class but every other class that depends upon it, directly or indirectly, and in a particular order. To make the determination about which classes need recompilation, the Java compiler maintains a directed graph of class dependencies. Any relationship in a UML class diagram of a Java program such as inheritance relationships, composition relationships and aggregations relationships indicate a class dependency.

The main class for this project P4GUI, should create the GUI shown below:



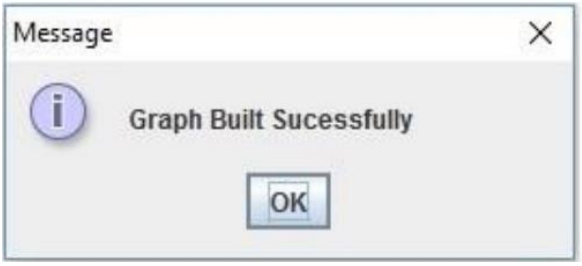
The GUI must be generated by code that you write. You may not use a drag-and-drop GUI generator.

Pressing the *Build Directed Graph* button should cause the specified input file that contains the class dependency information to be read in and the directed graph represented by those dependencies to be built. The input file should be generated by the students using a simple text editor such as Notepad. The input file associated with the above example is shown below:

```
ClassA ClassC ClassE
ClassB ClassD ClassG
ClassE ClassB ClassF ClassH
ClassI ClassC
```

Each line of this file specifies classes that have other classes that depend upon them. The first line, for example, indicates that `ClassA` has two classes that depend upon it, `ClassC` and `ClassE`. In the context of recompilation, it means that when `ClassA` is recompiled, `ClassC` and `ClassE` must be recompiled as well. Using graph terminology, the first name on each line is the name of a vertex and the remaining are its associated adjacency list of dependent classes. Classes that have no dependent classes need not appear at the beginning of a separate line. Notice, for example, that `ClassC` is not the first name on any line of the file.

After pressing the *Build Directed Graph* button, one of following two messages should be generated depending upon whether the specified file name could be opened:



Once the graph has been built, the name of a class to be recompiled can be specified and the *Topological Order* button can be pressed. Provided a valid class name has been supplied, a topological order algorithm should be executed that will generate (in the text area at the bottom of the GUI window) the list of classes in the order they are to be recompiled. The correct recompilation order is any topological order of the subgraph that emanates from the specified vertex. An invalid class name should generate an *InvalidClassNameException* custom exception causing an appropriate error message to be displayed in a *JOptionPane*. If the graph contains a cycle among the Java classes, a *GraphCycleException* custom exception will be thrown causing an appropriate error message to be displayed in a *JOptionPane*.
Note. In a real compiling processes, when circular dependencies exist in Java programs, the compiler must make two passes over all the classes in the cycle. For this program, it will be enough to display a message indicating that a cycle has been detected.

In addition to the main class that defines the GUI, a second class is needed to define the directed graph. It should be a generic class allowing for a generic type of its vertices. In this application the type of the vertices will be String. The graph should be represented as an array list of vertices that contain a linked list of their associated adjacency lists. The adjacency lists should be lists of integers that represent the index rather than vertex name itself. A hash map should be used to associate vertex names with their index in the list of vertices.

For the input file shown above the array list of linked lists of integers would be the following:

```
0 [1, 2]
1 []
2 [3, 6, 7]
3 [4, 5]
4 []
5 []
6 []
7 []
8 [1]
```

Storing the vertex indices rather than the names simplifies the topological order algorithm. The hash map would associate index 0 with `ClassA`, index 1 with `ClassC`, index 2 with `ClassE` and so on.

The directed graph class should define public methods for initializing the graph each time a new file is read in, for adding a vertex to the graph, for adding an edge to the graph and for

generating a topological order given a starting index.

Finally, custom checked exception classes should be defined for the cases where a cycle occurs and when an invalid class name is specified. Other classes / methods could be also defined to achieve the program requirements and/or to better structuring the code.
Your program should compile without errors.

You are to submit the following:

1. A `.zip` file that contains all the source code for the project (i.e. only `.java` files).
2. The solution description document `P4SolutionDescription` (.pdf or .doc / .docx) containing the following:
   a. Assumptions, main design decisions and error handling;
   b. A UML class diagram that includes all classes you wrote. Do not include predefined classes. You need only include the class name for each individual class, not the variables or methods;
   c. A table of test cases including the test cases that you have created to test the program. The table should have 5 columns indicating (1) what aspect is tested, (2) the input values, (3) the expected output, (4) the actual output and (5) if the test case passed or failed. Each test case will be defined in a separate table row.
   d. Relevant screenshots of program execution;
   e. Lessons learned from the project;

Grading Rubric:

Criteria	Meets	Does Not Meet
Design	**5 points**	**0 points**
	GUI is hand coded and matches required design (1)	GUI is generated by a GUI generator or does not match required design (0)
	Includes generic class for a directed graph (2)	Does not include a generic class a directed graph (0)
	Graph is represented as an array list of vertices that contain a linked list of their associated adjacency lists (1)	Graph is not represented as array list of vertices that contain a linked list of their associated adjacency lists (0)
	Includes custom checked exception classes for cycles and invalid class names (1)	Does not include custom checked exception classes for cycles and invalid class names (0)
Functionality	**10 points**	**0 points**
	Produces correct topological order for all cases without cycles (3)	Does not produce correct topological order for all cases without cycles (0)
	Produces error message for all cases with cycles (3)	Does not produce error message for all cases with cycles (0)
	Reports error message when file does not open (2)	Does not report error message when file does not open (0)
	Reports error message when invalid class name is entered (1)	Does not report error message when invalid class name is entered (0)
	Generates message confirming that graph has been built (1)	Does not generate message confirming graph has been built (0)
	5 points	**0 points**

Test Cases	Test cases include building and topological sort in a graph without cycles (2)	Test cases do not include building and topological sort in a graph without cycles (0)
	Test cases include a graph with cycles (1)	Test cases does not include a graph with cycles (0)
	Test cases include an invalid file name (1)	Test cases do not include an invalid file name (0)
	Test cases include an invalid class name (1)	Test cases do not include an invalid class name (0)
Documentation	**5 points**	**0 points**
	Solution description document includes project specific details for all the required, appropriate titled sections (1)	No solution description document is included (0)
	Source code follows Google recommendation Java style (1)	Source code does not follow Google recommendation Java style (0)
	Each source file includes a header of comments indicating (among others) file contents, the author and date (1)	Missing headers of the source files or failing to indicate the required information (0)
	Comment blocks of class description included with each class (1)	Comment blocks with class description not included with each class (0)
	Source code is commented and indented (1)	Source code is not commented and indented (0)
Overall Score	**Meets**	**Does not meet**
	16 or more	**0 - 15**