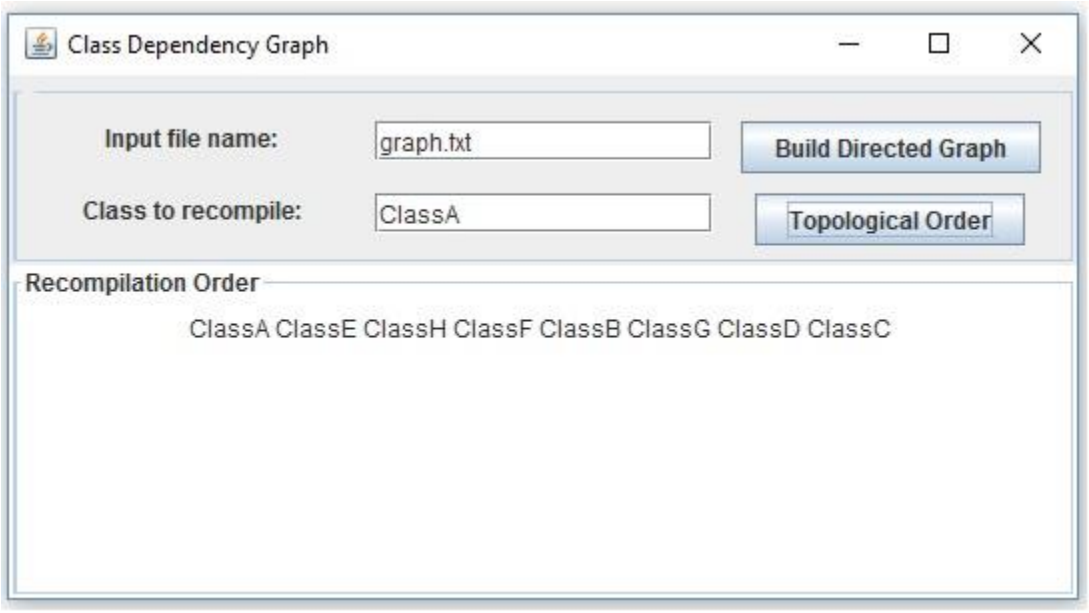
CMSC 350 Project 4

The fourth programming project involves writing 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 should create the GUI shown below:



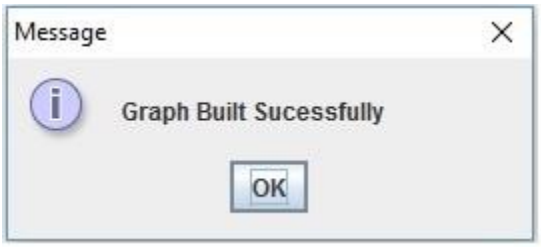
The GUI must be generated by code that you write. You may not use a drag-and-drop GUI generator.

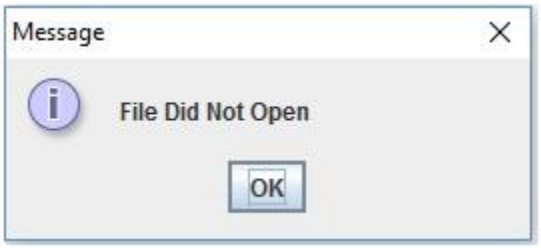
Pressing the *Build Directed Graph* button should cause the specified input file that contains the class dependency information to be read in and the directed graph represented by those dependencies to be built. The input file associated with the above example is shown below:

```
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 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. 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, the list of classes that need to be recompiled should be listed in the order they are to be recompiled in the text area at the bottom of the window. An invalid class name should generate an appropriate error message.

The correct recompilation order is any topological order of the subgraph that emanates from the specified vertex. Topological orders are not unique, but the one that is to be used for this program is the one generated using a depth-first search of the graph. The algorithm for generating this topological order is shown below:

```
depth_first_search(vertex s)
  if s is discovered
    throw cycle detected exception
  if s is finished
    return
  mark s as discovered
  for all adjacent vertices v
    depth_first_search(v)
  mark s as finished
  push s onto the stack
```

This algorithm generates a reverse topological order so after it completes, the forward topological order can be ascertained by popping the vertices off the stack. Note that an exception is to be thrown if the graph contains a cycle. When circular dependencies exist in Java programs, the compiler must make two passes over all the classes in the cycle, first compiling the specifications and subsequently the remaining code. For this program, it will be sufficient 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 for the vertex names. In this application those names will be strings. 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 depth-first search. The hash map would associate index 0 with `ClassA`, index 1 with `ClassC` and so on.

The directed graph class needs three public methods, one to initialize the graph each time a new file is read in, one to and an edge to the graph and one to generate a topological order given a starting index.

Finally checked exception classes should be defined for the cases where a cycle occurs and when an invalid class name is specified.

You are to submit two files.

1. The first is a `.zip` file that contains all the source code for the project, which includes any code that was provided. The `.zip` file should contain only source code and nothing else, which means only the `.java` files. If you elect to use a package the `.java` files should be in a folder whose name is the package name.
2. The second is a Word document (PDF or RTF is also acceptable) that contains the documentation for the project, which should include the following:
    a. 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
    b. A test plan that includes test cases that you have created indicating what aspects of the program each one is testing
    c. A short paragraph on 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 a generic class for a directed graph (2)	Does not include a generic class for 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 an array list of vertices that contain a linked list of their associated adjacency lists (0)
	Includes checked exception classes for cycles and invalid class names (1)	Does not Include 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 graph has been built (1)	Does not generate message confirming graph has been built (0)
Test Cases	**5 points**	**0 points**
	Test cases include a graph without cycles (2)	Test cases do not include 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**
	Correct UML diagram included (2)	Correct UML diagram not included (0)
	Lessons learned included (2)	Lessons learned not included (0)
	Comment blocks with class description included with each class (1)	Comment blocks with class description not included with each class (0)
Overall Score	**Meets**	**Does not meet**
	16 or more	**0-15**