In this program I tried to implement the ideas taught in class. From the basic Accessor methods for OOP to File I/O. The program I designed makes use of 2 classes to computes the route for the user to get from preferred source airport to the desired destination point.

The Node class created a Node with a unique source code, parent Node, the airline code that connects int to its parent and the number of stops between. In creating this class, I mastered accessibility modifiers. I made use of the private modifier to prevent the unique code from being exposed. To combat this and allow for the use of the values, I implemented getter methods for all the private attributes.

The Flight class, had separate methods to read from the files routes.csv, airports.csv, and userFile.txt. The main reason for this design was to allow for the flexibility in storing only the required material. Each file stored different records of different length and positions , therefore there was need to tailor each file reader differently. The *FileNotFoundExeption* was stated in the method signatures, but finally thrown and handled in the main method when the program was run. Additionally, to cater for users who may not be as conversant with exceptions, the method **userInfor()** returns a more user-friendly error message to hint the user to make adjustments to their file.

Now considering the routes, reading information from the routes.csv enabled generation of Node objects which were stored in a Hashmap<String, Nodes>. The parent attribute is intentionally set to null from the beginning. This is because no particular route is being generated yet. But when the user specifies their destination, the Nodes earn their parent attributes as they are generated from the frontier of the breadth first search.

The breadth first search occurs in the **generateRoute()** method. A bfs is then initiated to find the necessary flights to be combined to get to the desired destination. The first Node, the source, as no parent. But all the subsequent destination Nodes that it is connected to via certain airlines become child nodes to it. In the end, the **solution()** method is called to backtrack to the very first non-parent Node – thus the route would have been generated.

To print out the solution, I employed the Node *toString* method by overriding the original. The solution returns an *ArrayList<>* of Node that each provide the details below when printed :

Text

Description automatically generated

My solution provides optimality in terms of number of flights used to get to the destination by employing breadth first search. The Breadth first search works because it explores Nodes that are immediately after the sated origin. Therefore, it is almost guaranteed that the solution provided is optimal because it encounters the least costly (in terms of number of additional flights) airports first.