Assignment 1: Clicker Program.
Maciej Lis
cse13188, 211704558.

Program Design & How it works:

Note: Amongst the submitted files is a UML Diagram showing all the components and their relations. Comments within the code also provide a lot of insight on how the program works, and the overall design.

The program is separated into two standalone parts: Server and Client. The server application consists of;

- ClickerServer.java: the main application.
- ClickerProtocol.java: a seperate class, which keeps track of the clients progress throughout the session.
- ClickerMultiServerThread.java: threads that get spawned off for each new connection to the ClickerServer. This allows multiple simultaneous connections to our server.
- ClickerServerHelp.java: Holds various helpful functions, mostly related to file operations. This is a static class, so it doesn't need to be instantiated.
- class_list.txt: this is a line delimited list of all the students in a class.
- class_response.txt: the file where all responses are stored.

The Client:

- ClickerClient.java: a standalone application that uses the agreed upon protocol of the clicker client.

Server Description:
ClickerServer.java is the main class, which is run by the user (professer most likely). In order to continue to be able to receive input from the user when the questions have been initiated, we must run it in its own thread. The thread (thr1) creates a new ClickerMultiServerThread thread each time a client connects. This is what allows multiple connections to the server.

The ClickerMultiServerThread initiates a new instance of the protocol class *ClickerProtocol.java.* The ClickerProtocol class uses the state design pattern to keep track of the progress each client has made throughout their session. Using the state design pattern greatly increases the readability of the code, and allows for easy expansion of code. The protocol ends when a client successfully authenticates and answers a question - bye sending "Bye.". This keyword notifies the ClickerClient, to stop accepting input. Another keyphrase used by the protocol which notifies the ClickerClient application to not accept input is "Invalid student number. Connection closed.". So the connection closes when a client enters an invalid student number.

A list of valid student numbers are stored within the “class_list.txt” file, however its easy to change within the code by editing the class_list_file variable within ClickerServer.java. There are no hard-coded “magic constants” within the code.

Responses are stored within the “class_response.txt” file. Once again, this value is easy to change within the ClickerServer.java file. I chose to store responses in a text file, rather than in a database, primarily because of time restrictions. However some benefits of doing this are that, the professor can easily copy and paste the results, or copy the entire file without hassle, there are also less dependencies required for the application, and less overhead for running the program. This allows the program to easily be transferred to other systems. Some cons to using text files compared to text files: lack of encryption, speed, long term storage, and efficient parsing. But we can still use *grep* to easily search through our results.

I chose to implement ClickerServerHelp.java as a static class, because all the functions within it are standalone help functions, not requiring the instantiation of an object. Just like Javas Math class. By separating out all these functions from the other components of the Server application, we allow them to easily be reused by other parts, and by possible expansions we choose to implement in the future.

Difficulties arose when trying to stop the questions. Initially I tried to interrupt each of the threads, and within the ClientMultiServerThread check for this interrupt, and break out of the while loop. However this resulted in a very delayed stop to the server, allowing clients to finish up their current sessions, and in some cases even start new ones. My next attempt was to use the deprecated stop() function for threads, which worked - but the compiler warnings bothered me a lot. What I ended up doing was closing each socket instance within the clicker_clients_list (an ArrayList) which held a reference to each new ClientMultiServerThread thread instance. Than I closed the main socket. This worked well, stopping the clients immediately even if they were in the middle of an answering session.

A lot of extra features were added to the server, for example:

- *clear_file*: which will remove all the contents of the clicker response file.
- *num_connected*: output the current number of clients connected to the server. This helps the prof determine whether its a good time to stop the server. The way I did this was by simply iterating over the clicker_clients_list ArrayList, which help a reference to each ClientMultiServerThread, and checking if the thread is running.
- *is_running*: This will tell you if the start_question server is running. This is accomplished by checking if thr1 is running.

Client Description:

The client is a standalone application with no outside dependencies. This allows it to be easily transferred between systems. It may be run with or without command line options.

Proper error checking was kept in mind during the making of this program. All input fields are checked to make sure a valid input has been entered. Inputs are case insensitive, for the convenience of the user. “class_response.txt” file will be created if not found. I also used a lot of try-catch blocks.

Possible Improvements:

- command line options for the server. This will allow professors to start their servers significantly easier. By doing this, we essentially created an API to our application, allowing it to be run by third party applications that aren’t even written in Java.
- option for professor to change the class list and class response files within the program rather than altering the source code.
- GUI application for both the client and server application.
- use an actual database to store all responses and class lists.
- a more helpful help output.
- use “localhost” rather than having to type “127.0.0.1” in ClickerClient.
- show answer percentages. Example: 20% answered A, and 80% answered B.

Problematic Situations:

None to be reported.

How to run:

Compile:

```
javac *.java
```

Run server:

```
java ClickerServer
```

List of possible commands (case insensitive):

- START_QUESTIONS(n): start the clicker server. n = number of options.
- END_QUESTIONS: ends the clicker server, stopping connected clients from answering.
- LIST: reads the contents of the class_response_file, showing all the answers
- CLEAR_FILE: remove the contents of class_response_file
- NUM_CONNECTED: shows number of currently connected clients.
- IS_RUNNING: checks if the clicker server is running.
- EXIT: exit the application.

Run Client:

you can run multiple.

```
java ClickerClient 127.0.0.1 1337
```

or (will prompt you for server_IP and server_Port)

```
java ClickerClient
```