Sten Knutsen
CS352
PA1 Readme

ChatHistory Class

The ChatHistory class maintains a persistent chat history in a file named 'history.txt.' This file contains comma-separated values for the username's integer color value and the text of the chat itself. The *log()* method is used by the server to log public chat history information to the history file. *newRead()* reads the entire chat history into an array list and returns it to the server to be printed to the client. *read()* is a deprecated method (I used this in testing before introducing color-coded usernames).

Client Class

This class is based on the two example Client classes provided by the professor, but with some modifications. A second thread is generated so messages can be sent to and received from the server asynchronously. Client takes two command-line arguments: host address and port. If no server is running, Client keeps trying to connect to the server every three seconds.

ColorManagement Class

The ColorManagement class maintains a persistent database of usernames and their corresponding integer colors as comma-separated values in the file 'colors.txt.' The method *isInColorDatabase()* takes a username argument and returns true or false, depending on whether the given username is in the database file. *assignColor()* uses a random number generator to create an int color number that will be stored along with the username in the database. *getColorFromDatabase()* returns the int color number assigned to a given user in the database.

Server Class

This class is based upon the Server class example provided by the professor and like the Client class there are some significant additions. Most notable is the array of SessionThreads which manages all connections with clients. Also, messages regarding the establishment of client connections and warnings of a full chat session are published here. Server takes one command-line argument: port number.

SessionThread Class

SessionThread began as the example provided to our class but it has been significantly revised in order to exhibit all of the behaviors that the chat room described in the spec sheet demands.

First, SessionThread requires the user to enter their username using the **@name** command. It then validates the provided username, e.g. checks if name already active or if is greater than 100 characters. After the username is accepted, a color is assigned and the chat history is loaded and printed to the client. The new client's presence in the chatroom is announced to all users.

Next, the actual chat session loop starts. In addition to the commands required by the original assignment, I thought a couple other commands might be helpful to the user:

- **@pm** — tells user who is PM-ing them
- **@pwho** — tells user who they are currently PM-ing

Chat communications occur as outlined in the original spec sheet, with public messages being broadcast to all users and logged in history, and private messages being sent only to the targeted users. Public messages are displayed with a color-coded username, and private messages are not color-coded but tagged as private.

By the time we got around to talking about "locking" in class, I had already devised my own way of locking threads for the **@private** private messaging feature/command. Each session thread has a **pmSource** attribute which is initially set to null. When one user targets another for private messaging, the pmSource attribute is checked, and if null (person is not being PM'd by anyone else), private communication is instantiated by assigning the user name of the sender to pmSource. On the sender's side, the method *isPrivate()* iterates over all users and returns a boolean as to whether the sender is actively privately messaging at least one user. If so, all messages are sent to all and only users that have been targeted by the sender. When the sender uses the **@end** command, the targeted user's pmSource attribute is retuned to null. This "homegrown" method of locking ensures that while a sender is in private mode, all messages are sent only to targeted users, and that no other user may privately message users who are targets of private messaging.

Upon entering the **@quit** command, the client is notified that the connection is being terminated. All other members of the chat room are notified of the users departure, all active private messaging sessions are terminated, the user is removed from the session thread array and all connections are closed.

All throughout SessionThread, Java *synchronized statements* are employed whenever the sessionThreads array is accessed either to read or write information. Having these blocks of code contained within *synchronized* will (hopefully) coordinate access to the sessionThreads data structure and thus help preserve the integrity of the data therein.