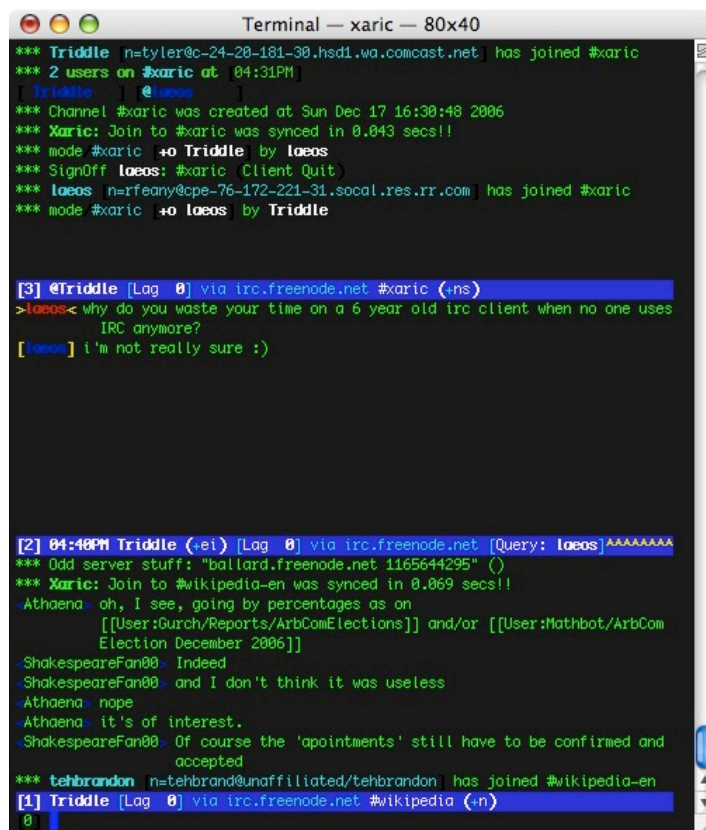
CSCI 136 Programming Exam #2
Fundamentals of Computer Science II
Spring 2014

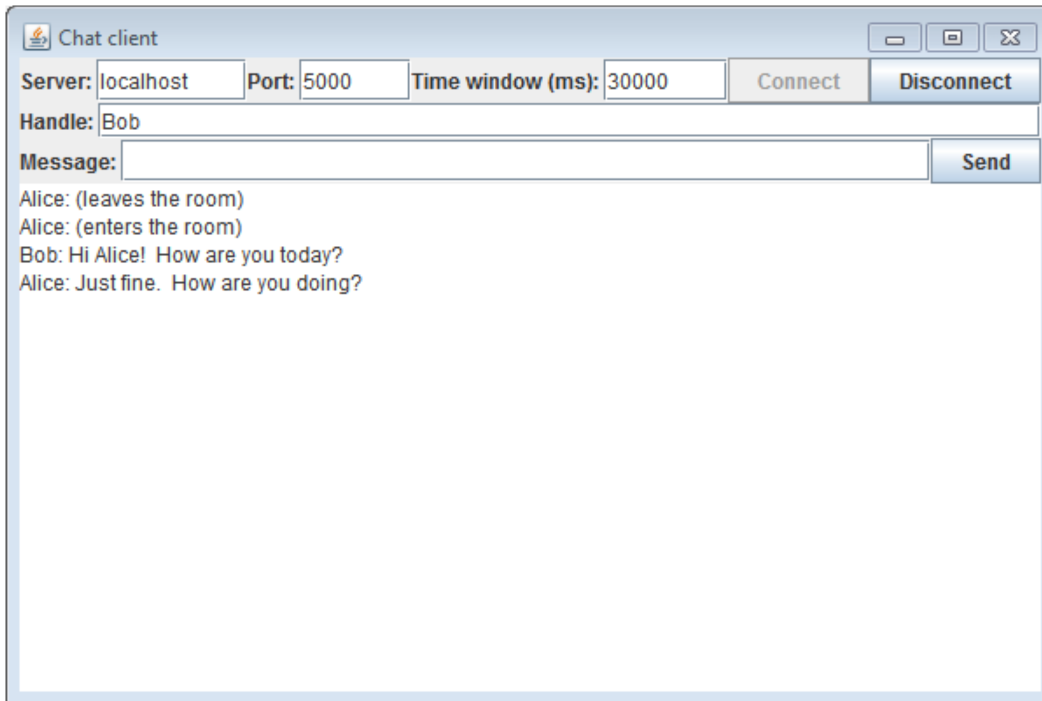
This part of the exam is like a mini-programming assignment. You will create a program, compile it, and debug it as necessary. This part of the exam is open book and open web. You may use code from the course web site or from your past assignments. When you are done, submit all your Java source files to the Moodle exam #2 dropbox. Please ***double check you have submitted all the required files***.

You will have 100 minutes. No communication with any non-staff members is allowed. This includes all forms of real-world and electronic communication.

Grading. Your program will be graded on correctness and to a lesser degree on clarity (including comments) and efficiency. You will lose a substantial number of points if your code does not compile or crashes on typical inputs.



Overview. You are building a multi-user text chat program. Each user in the chat program is identified by a handle. The client program periodically refreshes itself, updating the chat history displayed in the client. The user can configure how far in the past the chat history goes. Chat events are either a message sent by a user or an *event message* indicating a certain user has entered or exited the chat room. Here is a screenshot of the client:



Luckily another developer has come up with the GUI client. Your job is to implement the server-side support classes as well as the multi-threaded socket server program. Start by downloading the file at: http://katie.mtech.edu/classes/csci136/chat.zip

Part 1. Events are tracked by the server using the `Message` data type. A `Message` knows things like a `long` value indicating when the event occurred (the number of milliseconds since January 1st, 1970), a `String` handle of the user involved, and a `String` describing the message. Here is the API:

```
class Message
         Message(long timeStamp, String handle, String message)
 String toString()
boolean inTimeRange(long startTime, long endTime)
```

The `toString()` method should return a `String` consisting of the handle followed by a colon, a space, and then the message. For example, if a user with the handle `"Bob"` were to send the message `"Hello world!"`, `toString()` would return `"Bob: Hello world!"`.

The `inTimeRange()` method should return true if a message is in the time range `[startTime, endTime]`, inclusive of the end points, false otherwise. You can assume the start time given to this method is always less than or equal to the end time.

Part 2. Develop a class `Messages` that holds a collection of `Message` objects. You may use a data structure of your own choosing to store the collection (a Java built-in data type or your own, though we suggest using a built-in data type). Here is the API:

```
class Messages
---------------------------------------------------------------------
  void add(long timeStamp, String handle, String message)
String getInTimeRange(long startTime, long endTime)
```

As you might expect, the `add()` method simply adds a new message to the collection with the given timestamp, handle, and message.

The `getInTimeRange()` method returns a single line of text containing all messages that are in the given time range (inclusive of the start and end times). The messages should be in order from oldest to most recent. Messages on the line of text are separated from each other by a tab character. Tabs in Java can be generated using the special escape sequence `\t`. Each message is formatted per the `toString()` method in `Message`. Here is an example in which Alice and Bob have had a short conversation (the big spaced areas represent the tab separator):

```
"Alice: Welcome!       Bob: Thanks, how are you?       Alice: super groovy..."
```

Part 3. The main server program `ChatServer` is just boilerplate, it waits for a connection on a given port number and spins up a thread to handle the newly arriving client. You have been given a complete implementation of `ChatServer` which you should NOT need to modify.

Your job is to implement the `ChatServerWorker` class. This class handles a two-way socket conversation with a given client using a simple request-response protocol. Communication always starts with the client sending a command and possibly additional information to the server. A command is an uppercase `String` occurring by itself on a line of text. Here are the details of the protocol:

Command	Description
ENTER	The first command sent by a client when it initially connects. The client immediately follows the command with two additional lines of text: 1) The handle of the connecting user. 2) The connecting user's preferred time window in milliseconds (a long value). The server responds by sending a line of text containing all current messages that have occurred within the client's requested window. When a user connects, the server generates an *event message* informing all users of the new user. This event is just an automatically generated message attributed to the connecting user consisting of the text `"(enters the room)"`. The event generated by the ENTER command should be included in the server's response to the ENTER command that caused the event.
LEAVE	The last command sent by a client when it disconnects. The client immediately follows the command with one additional line of text: 1) The handle of the connecting user.

	The server responds by closing down the socket connection and the thread responsible for the client exits. There is no text sent back to the client in response to a LEAVE command. Similar to the ENTER command, when a user disconnects the server generates an event message informing everyone that the user has left. This event is just an automatically generated message attributed to the connecting user consisting of the text `"(leaves the room)"`.
MESSAGE	This command is used when the client sends a message to the other people in the chat room. The client immediately follows the command with two lines of text: 1) The handle of the user sending the message. 2) The text of the message. The server responds by returning a line of text containing all messages within the time window of the client (including the event message added as a result of this command).
UPDATE	This command is used by the client to periodically update its chat window. This is needed since we want clients to be able to see new messages without the user needing to send a new message. The command is NOT followed by any additional lines of text. The server responds by returning a line of text containing all messages within the time window of the client.

Hint 1: The server is responsible for determining the timestamp of each message. If you store messages in a sensible data structure, they will naturally be ordered by timestamp. In Java, you can obtain the number of milliseconds since January 1st, 1970 by calling the method `System.currentTimeMillis()`.

Hint 2: The `long` primitive in Java is just a bigger version of `int`. You can parse a `String` into a `long` using the `Long.parseLong()` method. If you need to specify a value of type `long`, you can use a cast or add an `L` to the end of the number, e.g. `long n = 42L;`

Hint 3: To ensure your socket communications don't get buffered and disrupt your protocol, be sure to set the `autoFlush` parameter to `true` when you construct your `PrinterWriter`.

All connected clients should see the same messages (subject to differences in the window time specified in the GUI client). For full credit, your server should be thread-safe, i.e. many clients should be able to connect simultaneously, exchanging messages without crashing and without losing any messages.

Feel free to add test `main()` methods to `Message` or `Messages` classes if you like (they will be ignored during grading). You may also add debugging statements to the `ChatClient` GUI program if you like. However, if you implement the server per the above protocol, it should work with the stock GUI client.

Submission. Submit your source files: `Message.java`, `Messages.java`, and `ChatServerWorker.java` to the Moodle dropbox.