CSC 330, Fall 2017
Lab 04

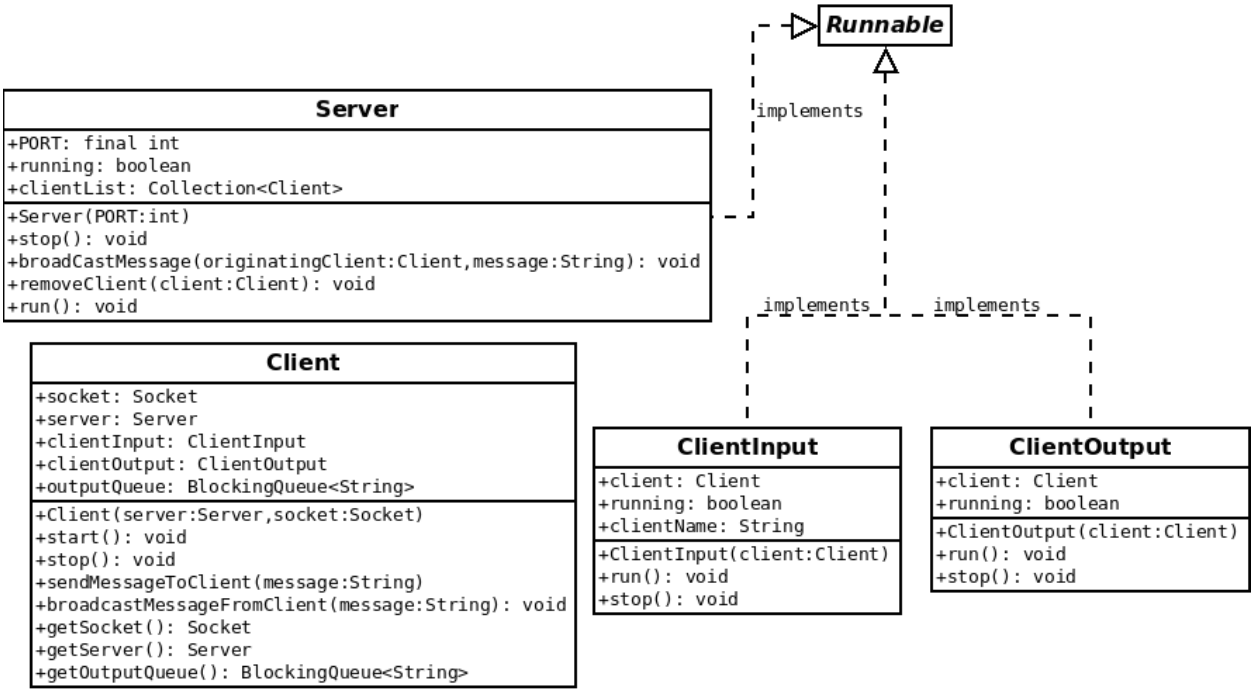
For Lab 4 our goal is to focus on the creation and management of a multithreaded network server. Our server is going to implement a *very basic* instant messenger functionality. When complete clients should be able to connect with a TCP socket to your server and for each carriage return delimited line of text sent to the server have that line of text be retransmitted to ALL other connected clients.

This lab will be due no later than 5:00 PM December 6th. Submission will consist of an electronic copy emailed to me with a subject that identifies it as your submission for Lab 4.

I do not have, nor am I requiring, any formal protocol for the communications between the clients and the server. The minimum functionality is that if I type "foo" in client a, while clients b and c are connected to the server each of clients b and c will display "foo" to their U.I. This will allow you to use a command line telnet client to test your server's functionality.

(Windows 10 users can enable telnet at the cmd prompt by pressing "Windows-Key" -> search for "Features" -> type "telnet", check the check box next to "Telnet Client", and then click ok)

For this assignment you are required to implement the following four classes as described in the UML and text below:



A full size version of this image is available at http://cs.rocky.edu/~bennera/Courses/csc330/Labs/2017/BasicInstantMessenger.png

The objects:

Server

The server's job is to accept incoming sockets, to act as a store of all of the connected clients, and to relay messages to the clients in it's clientList. The Server class implements the *Runnable* interface and as such will operate as the first thread in your solution. The run() method of your server must create a ServerSocket object bound to the PORT value specified during construction of the class, and then enter an infinite loop .accept()'ing new client connections. When a new client connection is accepted the server should add the client to the Collection<Client> list of clients, and then .start() the client. The details of the client's start() method are outlined below.

The methods of the Server are:

Server(PORT: final int):
: A simple constructor that allows the value of the PORT for this server to be set at construction time.

run() : void
: The run method as required by the *Runnable* interface. The run() method will handle accepting connections and creating Client instances as outlined above. The accept loop should continue while the running property of the server is true.

broadcastMessage(originatingClient: Client, message: String) : void
: This message is called by originatingClient and indicates that the server should dispatch the message to each other client in it's client list. A reference to the original client allows you to NOT retransmit the message back to the client that sends it.

removeClient(client: Client) : void
: When a Client object detects that the connection is no longer valid *(this will happen in the ClientInput object)* that Client must call the removeClient() method of the server so that the server stops trying to relay new messages to a non-existent client.

stop() : void
: The stop method is called from the invoking thread and should instruct each client to in turn stop() and then sets the server's running property to false.

Client

The Client object represents a single connected client and acts as a container for the two threads required to asynchronously perform input and output on the client's behalf.

The Client is *NOT RUNNABLE* and should not be made runnable. The purpose of the class is not to DO things it is to act as a container and coordinator for the actions of the ClientInput and ClientOutput threads.

The methods of the Client are:

Client(server: Server, socket: Socket) :

The constructor for the Client accepts a reference to the Server object constructing it as well as the connected Socket that the end user is communicating through. In addition to storing the passed properties the constructor is responsible for creating the BlockingQueue<String> used to coordinate output with ClientOutput object as well as instantiating both the ClientInput and ClientOutput classes. You should not create or start threads in the constructor.

start() : void
stop() : void

These two methods are used by the Server object to instruct a Client to start or stop it's contained threads for ClientInput and ClientOutput. The start() method should create and .start() threads for the ClientInput and ClientOutput objects, and the stop() method should just delegate to the .stop() method of the ClientInput and ClientOutput objects.

sendMessageToClient(message: String)

This method is used by the Server to indicate that a new message has been received and should be relayed to the end user. This method functions by placing the received message into the BlockingQueue where it will be picked up by the ClientOutput class. If you try to write the message directly it can cause problems in other clients threads.

sendMessageToClient(message) is called by the Server's broadcastMessage method!

broadcastMessageFromClient(message: String)

This method is used by the ClientInput object to indicate that the end user has sent a new line of text which should be relayed to all of the other clients. It should simply relay the passed message to the server's broadcastMessage() method.

broadcastMessageFromClient(message) is called by the ClientInput object.

getSocket() : Socket
getServer() : Server
getOutputQueue() : BlockingQueue<String>

These are simple getters leveraged by the ClientInput and ClientOutput objects. The just return the indicated property of the Client.

ClientInput

The ClientInput object implements the *Runnable* interface and is executed in a separate thread. This thread is responsible for reading data from the client via the connected Socket's input stream. Each time a carriage return ("\n") is read from the input it indicates that all previously read characters are a "line" of text and should be broadcast to all other clients.

The methods of ClientInput are:

ClientInput(client: Client) :

The constructor method should store the passed Client instance in a member variable and then initialize the running and clientName properties to true and null respectively.

stop() : void

The stop method simply set the running property to false indicating that the input loop should finish executing allowing the executing Thread to naturally terminate.

run() : void

This is the run() method required by the Runnable interface. The ClientInput's run() method should create a ByteArrayOutputStream to act as a variable length buffer and then enter a while() loop that continues while the running property is true and you can read more bytes from the socket: E.g.

```
while(running && ((ch = "read next byte from input stream") > 0)) {
        //…
}
```

If the input character is "\n" then the buffered bytes in the ByteArrayOutputStream should be converted to a string and passed to Client.broadcastMessageFromClient() and the buffer reset. If the character read is NOT a "\n" then it is added to the buffer.

My solution assumes that the first line of text sent by the client will be used as "clientName" and pre-pends that value to all output sent by the client. You are welcome to adopt this if desired. If you do not the clientName property won't make much sense and can be ignored / removed.

Because the .read() method of the InputStream provided by the socket is a blocking method it will cause the ClientInput object's execution thread to only execute when the end user has actually sent data.

If your while loop ends *for any reason* then you should in some way call the server's removeClient() method.

ClientOutput

The ClientOutput object manages writing data to the client. The problem is that we need to be able to coordinate and block output operations similarly to the ClientInput's read step. This is accomplished by using the BlockingQueue<String> contained in the Client. An instance of a BlockingQueue will cause an object calling the .take() method to block until data is available in the queue.

This is exactly the behavior we want in this case. The ClientOutput object can call .take() on the BlockingQueue and it will simply wait there until data becomes available. Once it has the data can be sent to the end user via the socket's OutputStream and we loop back around to waiting for the next thing.

The methods of ClientOutput are:

ClientOutput(client: Client) :

The ClientOutput constructor, similar to ClientInput, simply stores the passed Client object in a property and serves to initialize the private running property to true.

stop() : void

The stop method has a *tiny* bit more work to do in the ClientOutput object. Because our run method is going to spend most of its time blocked on .take() from the BlockingQueue it will not get to check the running property very often. For this reason to stop in a timely fashion you need to set running to false and then interrupt the thread executing the ClientOutput. This sounds mysterious but looks like this: Thread.currentThread().interrupt();
Doing so will cause the blocked .take() call to throw an InterruptedException (which can be ignored) and then return to the top of the loop to check running again.

run() : void

The run() method is prescribed by the *Runnable* interface here as well. This time we will simple execute a while loop that continues as long as the running property is true. Each iteration of the loop should .take() a single string from the Client's blockingQueue and if the message is not empty, write the bytes of the message to the socket's output stream. That's it. Single thread single purpose.

There are a NUMBER of opportunities to improve this project. A GUI that separates user input from server output. I protocol that better defines how the client is identified in the messages broadcast to the rest of the group. Think of the features of your favorite collaboration package (Slack, Discord, etc.) and take a whack at it.