Week 11 Lab - Recap: A Simple Chat System

COSC244

1 Assessment

This lab is worth 0.5%. The marks are awarded for the written answers to the questions. They should be stored electronically in a plain text file with a .txt suffix in your ∼/244/11 directory.

Make sure you get your work signed off by a demonstrator before leaving the lab.

2 A Simple Chat System

The purpose of this lab is to help you understand the code of *ChatServer.java*. It is one of the programs you can elect to implement during the practical test in Week 12. Understanding what is going on will greatly aid you in writing the program from scratch during the practical test.

In our chat system, whatever is typed on the keyboard of any client is sent to all clients (including the one that sent it). We do not type on the server's keyboard. Instead the server receives a line from a client and sends it to all clients. It outputs onto its screen information as clients connect and disconnect, and status information about what it is sending and to whom.

The code for our chat server is given below.

```java
// ChatServer.java

import java.io.*;
import java.net.*;
import java.util.*;

public class ChatServer {

   private static List<ClientHandler> clients = new LinkedList<ClientHandler>();

   public static void main(String[] args) {
      try {
         new ChatServer().startServer(Integer.parseInt(args[0]));
      } catch (Exception e) {
         e.printStackTrace();
         System.err.println("\nUsage: java ChatServer <port>");
      }
   }

```

```
  public void startServer(int port) throws Exception {
    ServerSocket serverSocket = new ServerSocket(port);
    System.err.println("ChatServer started");
    while (true) {
      ClientHandler ch = new ClientHandler(serverSocket.accept());
      System.err.println("Accepted connection from " + ch);
      synchronized (clients) {
        clients.add(ch);
      }
      ch.start();
    }
  }

  public static void sendAll(String line, ClientHandler sender) {
    System.err.println("Sending '" + line + "' to : " + clients);
    synchronized (clients) {
      for (ClientHandler cl : clients) {
        cl.send(sender + ": " + line);
      }
    }
  }

  public static class ClientHandler extends Thread {

    private BufferedReader input;
    private PrintWriter output;
    private String id;
    private static int count = 0;

    public ClientHandler(Socket socket) throws Exception {
      input = new BufferedReader(
          new InputStreamReader(socket.getInputStream()));
      output = new PrintWriter(socket.getOutputStream(), true);
      id = "client_" + ++count;
    }

    public void send(String line) {
      output.println(line);
    }

    public String toString() {
      return id;
    }

    public void run() {
      try {
        send("Welcome! You are " + this + ".");
        String line;
        while ((line = input.readLine()) != null) {
          sendAll(line, this);
        }
      } catch (IOException e) {
        e.printStackTrace();
      } finally {
        synchronized (clients) {
          clients.remove(this);
        }
        System.err.println(this + " closed connection!");
```

```
78            }
79         }
80      }
81  }
```

The class for this file is called *ChatServer* as given on line 7. There is also code for another class called *ClientHandler* declared on line 42. Line 9 creates a *LinkedList* of *ClientHandlers* called *clients*. This allows us to keep track of the clients.

Lines 11-18 are the *main()* method for the program. Line 13 creates a new *ChatServer* and calls the *startServer* method passing it the command line argument which is the port number. The remainder of the method is a *try-catch* block which you should understand.

Lines 20-31 are the *startServer()* method. Line 21 creates a new *ServerSocket* instance and assigns it to *serverSocket*. Line 22 prints a message about the server starting so we know something is going on.

Lines 23-30 form an infinite *while* loop. In line 24, we call the *accept()* method on *serverSocket*. The program waits until a client connects to the socket. When a client connects, *accept()* returns a *Socket* instance. That instance is passed to the *ClientHandler* constructor to create a new *ClientHandler* instance and assign it to *ch*. We'll look at the *ClientHandler* class soon.

Line 25 prints a message about accepting a connection and gives the client's identification. Lines 26-28 add the client to the *clients* list. Since multiple threads can access *clients* at unpredictable times, we use the *synchronized()* keyword to assure exclusive access. Line 27 adds the new client to the list.

Finally, line 29 calls *start()* to start the new *ClientHandler* thread executing.

Lines 33-40 are the *sendAll()* method of *ChatServer*. This method has two formal parameters: a *String* and a *ClientHandler*. The first parameter, *line*, is the *String* to be sent. The second, *sender* is the client which sent the *String* to the server.

Line 34 prints a status message on the server's screen about sending a message to the clients. Refer to your answers from Lab 9 for the form of this message. That will help you understand the argument to *System.err.println()*.

Lines 36-38 are a *for* loop on *clients* which sends the message to each client by calling the *send()* method of *ClientHandler* once for each client. Lines 35 & 39 give the method exclusive access to *clients*.

Lines 42-80 are the code for the *ClientHandler* class. There is an instance of this class for each client that connects to the server. Private variables are declared in lines 44-47.

Lines 49-54 are the class constructor which is passed a *Socket* instance. Lines 50 & 51 get an *InputStream* on the socket, wrap a *InputStreamReader* around it, wrap a *BufferedReader* around that, and finally assign the whole mess to *input*. Line 52 gets an *OutputStream* on the socket, wraps a *PrintWriter* around it and assigns the result to *output*. Line 53 creates an *id String* so the client can be identified.

Lines 56-58 are the *send()* method of *ClientHandler*. Its purpose should be obvious. Lines 60-62 are the *toString()* method. Again the purpose should be obvious.

Lines 64-79 are the *run()* method of the class. It is called whenever a *ClientHandler* thread is started. It is a *try-catch-finally* block.

Line 66 causes a message to be printed in the clients window giving its identity. Lines 68-70 form a *while* loop that repeatedly reads a line from *input* and calls the *sendAll()* method of *ChatServer*. In this case, the input is the socket of the thread. The result is that whatever the connected client sends is printed on the screen of all clients.

The *finally* part of the *try-catch-finally* block is executed as the thread terminates. For us, this normally occurs when you type ctrl-C in a client's window. Lines 74-76 remove the terminated client from the *clients* list, using exclusive access to *clients*. Line 77 prints a message stating which client closed.

3 Test Your Understanding

Test your understanding of *ChatServer* by answering the following questions. You can do this either as preparation before the lab or during your lab slot.

Q1 Explain the purpose of the *send()* method of *ClientHandler*.

Q2 Where is the *line* in *send()* going to?

Q3 Explain the purpose of the *toString()* method of *ClientHandler*.

Q4 When a client closes a connection and terminates, and then a new client connects, is the client number recycled? Why or why not?

Q5 List the lines of code executed when the *ChatServer* starts up to the point where it is listening for its first connection. The list should be in execution order.

Q6 If 3 clients have connected to *ChatServer*, how many threads are running assuming no client has terminated. Include the client threads in your count.

Q7 Describe what each thread from **Q6** is doing. For threads which are performing the same task but for different clients, you should state how many of that particular thread is running. If you did not correctly answer **Q6**, complete this question after you get the correct answer and before you leave the lab. You will not get the mark for this lab until both questions are answered correctly.