|  |  |
| --- | --- |
|  | 소켓 프로그래밍에서 멀티스레드의 사용법과 server-client 프로그래밍을 이해한다 |

**Requirements**

* < 요구조건 #1: 멀티스레드를 이용해서 여러 클라이언트를 한번에 다룬다>
* < 요구조건 #2: [whisper]을 사용해서 1대1 채팅이 가능하게 한다(기존의 코드는 broadcasting 채팅)>
* < bouns feature: [whisper]을 gui를 이용해 구현한다>

# REPORT

서버를 먼저 실행하고 클라이언트를 실행하면 채팅창 앞에 이름을 입력하는 창이 뜨게 된다.

클라이언트를 여러 번 실행하면 원하는 만큼 채팅창을 열 수 있다.

이름을 입력한 후 확인 버튼을 누르면 입력한 이름이 서버로 전달되고, 이름을 입력하지 않거나 취소버튼을 누르면 null이 서버로 전달된다.

이름 입력 후 확인/취소 버튼을 누르면 채팅창이 뜨게 된다.

채팅창 아래에 메시지를 입력하는 란이 있다.

메시지 입력란에 문자열을 입력하면 현재 열려있는 채팅창에 모두 문자열이 전달된다.

메시지 입력란에 <이름/> 이러한 형식과 보낼 메시지를 입력하면 ‘이름’에 해당하는 채팅창에만 문자열이 전달된다.

## SOURCE CODES

ChatClient

|  |
| --- |
| **package** myPackage;  **import** java.awt.event.ActionEvent;  **import** java.awt.event.ActionListener;  **import** java.io.IOException;  **import** java.io.PrintWriter;  **import** java.net.Socket;  **import** java.util.HashMap;  **import** java.util.Scanner;  **import** java.awt.BorderLayout;  **import** javax.swing.JButton;  **import** javax.swing.JComboBox;  **import** javax.swing.JFrame;  **import** javax.swing.JOptionPane;  **import** javax.swing.JScrollPane;  **import** javax.swing.JTextArea;  **import** javax.swing.JTextField;  /\*\*  \* A simple Swing-based client for the chat server. Graphically it is a frame with a text  \* field for entering messages and a textarea to see the whole dialog.  \*  \* The client follows the following Chat Protocol. When the server sends "SUBMITNAME" the  \* client replies with the desired screen name. The server will keep sending "SUBMITNAME"  \* requests as long as the client submits screen names that are already in use. When the  \* server sends a line beginning with "NAMEACCEPTED" the client is now allowed to start  \* sending the server arbitrary strings to be broadcast to all chatters connected to the  \* server. When the server sends a line beginning with "MESSAGE" then all characters  \* following this string should be displayed in its message area.  \*/  **public** **class** ChatClient{  String serverAddress; //서버의 주소  Scanner in; //서버로부터 전송된 스트림을 받는 변수  PrintWriter out; //서버에 문자열을 전송하는 함수  JFrame frame = **new** JFrame("Chatter"); //채팅창  JTextField textField = **new** JTextField(50); //메세지를 입력받는 공간  JTextArea messageArea = **new** JTextArea(16, 50); //입력된 메세지를 보여주는 공간      /\*\*  \* Constructs the client by laying out the GUI and registering a listener with the  \* textfield so that pressing Return in the listener sends the textfield contents  \* to the server. Note however that the textfield is initially NOT editable, and  \* only becomes editable AFTER the client receives the NAMEACCEPTED message from  \* the server.  \*/  **public** ChatClient(String serverAddress) {  **this**.serverAddress = serverAddress;  textField.setEditable(**false**);  messageArea.setEditable(**false**);  frame.getContentPane().add(textField, BorderLayout.***SOUTH***);  frame.getContentPane().add(**new** JScrollPane(messageArea), BorderLayout.***CENTER***);  frame.pack();  // Send on enter then clear to prepare for next message  textField.addActionListener(**new** ActionListener() {  **public** **void** actionPerformed(ActionEvent e) {  out.println(textField.getText());  textField.setText("");  }  }    );  }  **private** String getName() { //이름을 입력받도록 하는 창이 뜨게 하는 함수  **return** JOptionPane.*showInputDialog*(  frame,  "Choose a screen name:",  "Screen name selection",  JOptionPane.***PLAIN\_MESSAGE*** //이름을 입력받고 입력값이 없으면 null 반환  );  }  **private** **void** run() **throws** IOException {  **try** {  Socket socket = **new** Socket(serverAddress, 59001); //소켓을 생성한다  in = **new** Scanner(socket.getInputStream());//서버로부터 전송되어서 온 스트림을 읽는다  out = **new** PrintWriter(socket.getOutputStream(), **true**);//서버에 문자열을 보낸다    **int** i=0;    **while** (in.hasNextLine()) { //전송되어 온 스트림이 없을 때 까지 반복한다  String line = in.nextLine(); //전송되어온 스트림을 line에 저장한다  **if** (line.startsWith("SUBMITNAME")) { //전송되어온 스트림이 SUBMITNAME으로 시작한다면  out.println(getName()); //getName함수(클라이언트 이름을 입력받는 함수)를 호출한다    } **else** **if** (line.startsWith("NAMEACCEPTED")) { //전송되어온 스트림이 NAMEACCEPTED으로 시작한다면  **this**.frame.setTitle("Chatter - " + line.substring(13)); //채팅창 맨 위에 Chatter-입력한 이름이 뜨도록 한다  textField.setEditable(**true**); //메세지를 입력받는 공간을 클라이언트가 쓸 수 있도록 한다  } **else** **if** (line.startsWith("MESSAGE")) { //전송되어온 스트림이 MESSAGE로 시작한다면  messageArea.append(line.substring(8) + "\n"); //입력된 메시지가 채팅창에 보이게 한다  }  }  } **finally** {  frame.setVisible(**false**); //채팅창을 보이지 않게 한다  frame.dispose(); //채팅창을 없앤다  }  }  **public** **static** **void** main(String[] args) **throws** Exception {    ChatClient client = **new** ChatClient("127.0.0.1"); //서버의 주소를 전달한다  client.frame.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***); //채팅창의 x표시를 누르면 채팅창이 닫히게 한다  client.frame.setVisible(**true**); //채팅창이 보이게 한다  client.run(); //run함수를 실행한다  }  } |

ChatServer

|  |
| --- |
| **package** myPackage;  **import** java.io.IOException;  **import** java.io.PrintWriter;  **import** java.net.ServerSocket;  **import** java.net.Socket;  **import** java.util.Set;  **import** java.util.HashMap;  **import** java.util.HashSet;  **import** java.util.Scanner;  **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;  /\*\*  \* A multithreaded chat room server. When a client connects the server requests a screen  \* name by sending the client the text "SUBMITNAME", and keeps requesting a name until  \* a unique one is received. After a client submits a unique name, the server acknowledges  \* with "NAMEACCEPTED". Then all messages from that client will be broadcast to all other  \* clients that have submitted a unique screen name. The broadcast messages are prefixed  \* with "MESSAGE".  \*  \* This is just a teaching example so it can be enhanced in many ways, e.g., better  \* logging. Another is to accept a lot of fun commands, like Slack.  \*/  **public** **class** ChatServer{  // All client names, so we can check for duplicates upon registration.  **private** **static** Set<String> *names* = **new** HashSet<>(); //클라이언트가 입력한 중복되지 않은 이름을 저장한다  // The set of all the print writers for all the clients, used for broadcast.  **private** **static** Set<PrintWriter> *writers* = **new** HashSet<>(); //클라이언트에게 문자열을 전송하는 중복되지 않은 변수를 저장한다    **private** **static** HashMap<String, PrintWriter> *hash* = **new** HashMap<String, PrintWriter>(); //이름에 해당하는 printwriter type 변수를 저장(whisper에서 사용)    **static** ExecutorService *pool* = Executors.*newFixedThreadPool*(500); //여러 클라이언트를 위한 멀티스레드    **public** **static** **void** main(String[] args) **throws** Exception {  System.***out***.println("The chat server is running..."); //콘솔창에 서버가 시작됬다는 것을 출력한다  //ExecutorService pool = Executors.newFixedThreadPool(500); //스레드 500개를 생성해서 재사용한다  **try** (ServerSocket listener = **new** ServerSocket(59001)) { //서버 소켓을 생성한다  **while** (**true**) {  *pool*.execute(**new** Handler(listener.accept(), *hash*)); //스레드에서 클라이언트 접속 요청 대기 및 허락  }  }  }  /\*\*  \* The client handler task.  \*/  **private** **static** **class** Handler **implements** Runnable {  **private** String name;//입력한 이름을 저장하는 변수  **private** Socket socket; //소켓 변수  **private** Scanner in; //전송되어진 스트림을 받는 변수  **private** PrintWriter out; //문자열을 전송하는 변수    String usernm; //whisper에서 전송할 이름 저장 변수  HashMap<String, PrintWriter> hash; //이름과 이름에 보낼 printwriter를 mapping  /\*\*  \* Constructs a handler thread, squirreling away the socket. All the interesting  \* work is done in the run method. Remember the constructor is called from the  \* server's main method, so this has to be as short as possible.  \*/  **public** Handler(Socket socket, HashMap hash) {  **this**.socket = socket;  **this**.hash = hash; //whisper를 위해 추가  }  /\*\*  \* Services this thread's client by repeatedly requesting a screen name until a  \* unique one has been submitted, then acknowledges the name and registers the  \* output stream for the client in a global set, then repeatedly gets inputs and  \* broadcasts them.  \*/  **public** **void** run() {    **try** {  in = **new** Scanner(socket.getInputStream()); //클라이언트가 전송한 스트림을 받는다  out = **new** PrintWriter(socket.getOutputStream(), **true**); //클라이언트에게 문자열을 보낸다    // Keep requesting a name until we get a unique one.  **while** (**true**) {  out.println("SUBMITNAME"); //SUBMITNAME을 클라이언트에게 전송한다  name = in.nextLine(); //클라이언트에게서 받은 문자열을 이름 변수에 저장한다  **if** (name == **null**) { //클라이언트에게서 받은것이 없으면 반복문을 중단한다  **return**;  }  **synchronized** (*names*) { //names가 다른부분에서는 실행되지 않도록 점유한다(동기화)  **if** (name.length() > 0 && !*names*.contains(name)) { //name에 저장된 값이 있고 이미 입력된 이름이 아니면  *names*.add(name); //names hashset에 이름을 저장한다  hash.put(name,out); //hashmap에 이름과 클라이언트로 문자열을 전소할 수 있는 out을 저장한다  **break**;  }  }  }      // Now that a successful name has been chosen, add the socket's print writer  // to the set of all writers so this client can receive broadcast messages.  // But BEFORE THAT, let everyone else know that the new person has joined!  out.println("NAMEACCEPTED " + name); //클라이언트에서 전달된 이름을 받았다고 클라이언트에게 알려준다  **for** (PrintWriter writer : *writers*) {  writer.println("MESSAGE " + name + " has joined"); //채팅창에 클라이언트 쪽에서 입력한 이름이 참가했다고 출력한다    }  *writers*.add(out); //클라이언트로부터 연결요청에 의해 생성된 소켓을 이용해 출력스트림을 생성하여 출력객체를 만든다        // Accept messages from this client and broadcast them.  **while** (**true**) {  String input = in.nextLine(); //클라이언트로 받은 스트림을 저장한다  **if** (input.toLowerCase().startsWith("/quit")) { //quit으로 시작한다면 반복문 탈출  **return**;  }    //whisper  **if** (input.toLowerCase().startsWith("<")) { //<로 시작한다면 whisper일 수도 있다  usernm = input.substring(input.indexOf("<")+1, input.indexOf("/")); //whisper 대상의 이름만 추출한다  String msgs = input.substring(input.indexOf(">")+1); //whisper의 메세지만 추출한다  **if**(*names*.contains(usernm)){ //이전에 입력된 이름 리스트에 whisper 대상의 이름이 저장되어있다면  PrintWriter whisper = hash.get(usernm); //whisper 대상의 이름에 해당하는 printwriter를 가져오고  whisper.println("MESSAGE " + name + ": " + msgs); //추출한 문자만 전송한다  }  }  **else**{  **for** (PrintWriter writer : *writers*) { //모든 출력객체를 통해 broadcasting 한다  writer.println("MESSAGE " + name + ": " + input); //메세지 보낸 이름과 문자 내용을 클라이언트에게  }  }      }  } **catch** (Exception e) {  System.***out***.println(e);  } **finally** {  **if** (out != **null**) {  *writers*.remove(out); //출력객체에서 삭제한다  }  **if** (name != **null**) {  System.***out***.println(name + " is leaving");  *names*.remove(name); //이름저장set에서 삭제한다  **for** (PrintWriter writer : *writers*) {  writer.println("MESSAGE " + name + " has left"); //모두에게 나간사람이 있다는 것을 채팅창에 출력한다  }  }  **try** { socket.close(); } **catch** (IOException e) {}  }  }  }  } |

## OUTPUT (Screen Shots)







