# Threading Chat Server

# The Result

```
Please enter username:
jill
Welcome to the chat, jill
tom: Hi guys
Jack: Whats up
Heyyy
jill: Heyyy
tom: Imma get the homework done an be back in 10
Another client has disconnected
Jack: By then i guess
```

Multiple clients can join and have the ability to:

- Create a username
- Send and receive messages to/from other connected clients
- Leave the chat on ‘Exit’

# Threading

Task focused on threading

Concurrently running:

- Observer → `public class MessageDisplay { //Observer to hold current message for broadcasting and a list of all the clients`
- Handler for the clients → `public class ClientHandler extends Thread {//Thread to deal with incoming client messages etc.`
- Inputs for server → `public class ServerInputThread extends Thread { //Deals with user input into the console,`
- Clients sending → `public class WriteThread extends Thread { //Deals with scanning and sending any messages typed in to the clients console by user`
- Clients receiving → `public class ReadThread extends Thread { //Thread to read incoming communications to client`

# The Server

- Server port can be specified in console or uses default 14001.

```
public static void main(String[] args) throws IOException {
    int i;
    int port = 14001;
    for (i=0;i<args.length;i++){ //If there are no parameters passed in, will not execute.
        if (args[0].equals("-csp")){ //If the parameter passed in is valid, the proceeding value is set as new port
            port = Integer.parseInt(args[1]);
        }
    }
    System.out.println("Server listening");
```

- Initialise ***Observer***, ***Socket*** and ***Server input handler***

```
MessageDisplay messagedisplay = new MessageDisplay(); //Creating new observer object
ServerSocket in = new ServerSocket(port); //Initialising socket with given or default port
Thread sit = new ServerInputThread(in, messagedisplay); //Creating ServerInputThread
sit.start();
```

- Attempt connection of socket and por initialise input/output streams, start handler for clients.

```
try {
    s = in.accept();
    System.out.println("Server accepted connection on " + in.getLocalPort() + " ; " + s.getPort());
    DataInputStream dis = new DataInputStream(s.getInputStream()); //Creating data output and input stream
    DataOutputStream dos = new DataOutputStream(s.getOutputStream());
    Thread t = new ClientHandler(s, dis, dos, messagedisplay); //Creating ClientHandler thread
    t.start(); //Must be a thread because multiple clients need to connect
```

# The Server: Inputs

Required to be able to manually close the server safely.

```
public class ServerInputThread extends Thread { //Deals with user input into the console, mu
    Scanner scn = new Scanner(System.in); //Initialising
    private ServerSocket in;
    private MessageDisplay messagedisplay;


    public ServerInputThread(ServerSocket in, MessageDisplay messagedisplay) { //Constructor
        this.in = in;
        this.messagedisplay = messagedisplay;
    }


    public void run() {
        while (true) { //Runs indefinitely
            try {
                String isExit = scn.nextLine(); //For console input
                if(isExit.equals("Exit")){
                    System.out.println("Server Closing");
                    messagedisplay.setMessage("Server Has Closed, terminating connection");
                    this.in.close(); //closes ServerSocket
                    System.exit( status: 0); //terminate program
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

# The Client Handler

- Initialises the input/output streams, socket and observer in constructor.

```
public ClientHandler(Socket s, DataInputStream dis, DataOutputStream dos, MessageDisplay messagedisplay) { //Constructor for ClientHandler
    this.s = s;
    this.dis = dis;
    this.dos = dos;
    this.messagedisplay = messagedisplay;
    this.messagedisplay.attach( clienthandler: this); //This attaches this clienthandler object to the arraylist in the observer
}
```

- Reads what this handler's client sends, adds it to observer

```
while (true) {
    received = dis.readUTF(); //Reads incoming data for input from its client
    if (!received.equals("Exit")) { //If any message that isn't exit
        messagedisplay.setMessage(received); //Sets state of observer's message to what the client sent
```

- Removes client from observer if necessary

```
else{ //If the client typed exit
    messagedisplay.setMessage("Another client has disconnected");
    messagedisplay.disattach( clienthandler: this); //Removes it from observer's arraylist
```

- Writes the current message in observer to client

```
public void update() {
    try {
        dos.writeUTF(messagedisplay.getMessage()); //Sends the current message held by the observer to this handlers specific client
```

# The Observer

Contains list of clients, can return or change the message it is holding and updates the client handlers on a new input

```java
public class MessageDisplay { //Observer to hold current message for broadcasting and a list of all the clients to broadcast to.

    private List<ClientHandler> clients = new ArrayList<~>(); //Creating list to hold client handlers (individual clients)
    private String message;

    public String getMessage() { return message; }

    public void setMessage(String message) { //Takes incoming message and sets as most recent one
        this.message = message;
        updateAllClients(); //When new message is received it needs to be broadcast to all clients
    }

    public void attach(ClientHandler clienthandler) { clients.add(clienthandler); }

    public void disattach(ClientHandler clienthandler) { clients.remove(clienthandler); }

    public void updateAllClients(){ //Cycles through the list of client handlers and runs their update methods
        for (ClientHandler clienthandler : clients) {
            clienthandler.update(); //Sends the message to the client via client handler to be output on console
        }
    }
}
```

# The Client

- Takes in the server port (default 14001 again) and the clients own host (default localhost).

```
public static void main(String[] args){
    try {
        int i;
        int port = 14001; //default values for port and host
        String host = "localhost";
        for (i=0;i<args.length;i++){ //If parameters have been passed in
            if (args[i].equals("-cca")){ //if '-cca' then the following argument is used as host
                host = (args[i+1]);
            }
            else if (args[i].equals("-ccp")){ //if '-ccp' then the following argument is used as port
                port = Integer.parseInt(args[i+1]);
            }
        }
        Socket s = new Socket(host, port); //Creating objects for socket and data streams and scanner
```

- Sets up input/output streams from server

```
DataInputStream dis = new DataInputStream(s.getInputStream());
DataOutputStream dos = new DataOutputStream(s.getOutputStream());
```

- Takes user specified username

```
Scanner scn = new Scanner(System.in);
System.out.println("Type Exit at any time to disconnect the client.\nPlease enter username:");
String username = scn.nextLine(); //Username must be entered before any other actions occur so doesn't need to be thread
System.out.println("Welcome to the chat, " + username);
```

- Starts the necessary read/write threads

```
Thread r = new ReadThread(dis); //Creating read and write threads
Thread w = new WriteThread(s, dos, dis, username);
w.start();
r.start();
```

# The Client: Read Thread

Simply takes incoming data in unicode format and prints it to the user.
If message is a specific one from the server, terminates client.

```
public class ReadThread extends Thread { //Thread to read incoming communications to client
    private DataInputStream dis;



    public ReadThread(DataInputStream dis) { this.dis = dis; }


    public void run() {
        while (true) { //Runs indefinitely
            try {
                String received = dis.readUTF(); //Reads incoming data stream in UTF (Unicode Transformation Format)
                System.out.println(received); //prints any messages received to the clients console
                if (received.equals("Server Has Closed, terminating connection")){ //If message is the termination message from the server
                    System.exit( status: 0); //End the client program
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

# The Client: Write Thread

- Takes user input, sends to client handler, which will then send it to the observer

```
else if ((!toSend.equals("Exit")) && (permission == 0)) { //If a normal message
    dos.writeUTF( str: username + ": " + toSend); //Writes to the clients handler
}
```

- Does not permit user to send server shutdown message

```
String toSend = scn.nextLine(); //Read console input
if(toSend.equals("Server Has Closed, terminating connection")){ //Prevents user from sending termination
    System.out.println("Sorry, you do not have permission to send that message");
    permission = 1; //Sets permission to 1 to stop message from sending in next line
}
```

- Disconnects user on input ‘Exit’

```
else {
    if(permission == 0) { //If message is Exit
        System.out.println("Exiting");
        dos.writeUTF(toSend); //Notifies handler of exiting so it can disattach from observer
        this.dos.close(); //Closes Data streams and socket
        this.dis.close();
        this.socket.close();
        System.exit( status: 0); //Terminates program
    }
```