1. Introduction

Socket programming enables communication between applications over a network. This documentation describes a Java-based multi-client chat application that uses TCP sockets and threading to allow multiple clients to communicate simultaneously. A graphical user interface (GUI) is implemented using Java Swing.

2. Objectives

- To understand TCP socket programming in Java
- To implement multi-client communication using threads
- To design a GUI-based chat application using Java Swing
- To demonstrate real-time message exchange

3. Tools and Technologies

- Java JDK 8 or above
- Java Swing
- TCP/IP Socket Programming
- IntelliJ IDEA / VS Code

4. System Description

The system consists of one server and multiple clients. The server listens for incoming connections and creates a separate thread for each connected client. Messages sent by one client are broadcast to all other clients. Each client uses a graphical user interface to send and receive messages.

5. Source Code

for Class ChatServer.Java

```
1   import java.io.*;
2   import java.net.*;
3   import java.util.*;
4
```

1.import java.io.*;

This imports **input/output classes**. And used for

- Reading data (files, keyboard, socket input)
- Writing data (files, socket output)

2.import java.net.*;

This imports **networking classes**. And used for

- Client–server communication

- Working with IP addresses and ports

`3.import java.util.*;`

This imports **utility classes**. And used for

- Data structures
- Helpers like scanners and timers

```java
public class ChatServer {
    private static final int PORT = 5000; // same port as your ChatGUI
    private static Set<PrintWriter> clientWriters = new HashSet<>();

    Run main | Debug main
    public static void main(String[] args) {
        System.out.println("Chat Server started on port " + PORT);

        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("New client connected: " + clientSocket.getInetAddress());

                // Start a new thread to handle this client
                new Thread(new ClientHandler(clientSocket)).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

1️⃣ `private static final int PORT = 5000;`

- This sets the **port number** the server will listen on.
- `5000` is just a number; both client and server must use the **same port** to communicate.
- `final` means this value **cannot change**.
- `static` means it belongs to the class, not an instance.

2️⃣ `private static Set<PrintWriter> clientWriters = new HashSet<>();`

- This is a **set of writers** for all connected clients.
- `PrintWriter` is used to **send messages to clients**.
- `Set` ensures there are **no duplicate writers**.
- This allows the server to **broadcast messages** to all clients.

3️⃣ `ServerSocket serverSocket = new ServerSocket(PORT);`

- ☐ `ServerSocket` is a special socket that **listens for incoming client connections**.
- ☐ The server will **wait for clients** on the specified port.

4️⃣ `serverSocket.accept();`

- ☐ This **blocks** (waits) until a client tries to connect.
- ☐ When a client connects, it returns a **Socket object** representing that client.
- ☐ `clientSocket.getInetAddress()` gets the client's IP address.

5️⃣ `new Thread(new ClientHandler(clientSocket)).start();`

- ☐ Each client is handled **in a separate thread** so multiple clients can chat **at the same time**.
- ☐ `ClientHandler` is a class (you probably have it in your code) that:
  - ☐ Reads messages from the client
  - ☐ Sends messages to other clients
- ☐ `start()` begins the thread.

6️⃣ The infinite loop `while(true)`

- ☐ Keeps the server **running continuously**.
- ☐ Accepts new clients as they connect.

7️⃣ `try` and `catch (IOException e)`

- ☐ Handles errors like:
  - ☐ Port already in use
  - ☐ Connection problems
- ☐ `e.printStackTrace();` prints the error to the console.

```java
private static class ClientHandler implements Runnable {
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;

    public ClientHandler(Socket socket) {
        this.socket = socket;
    }
```

1️⃣ `private static class ClientHandler implements Runnable`

- This defines a **nested class** inside your server class.
- `implements Runnable` means this class can be run in a **separate thread**.
- Each client connection will have **its own ClientHandler thread**, so multiple clients can chat simultaneously.

2️⃣ `private Socket socket;`

- Stores the **client's socket connection**.
- The socket is used to:
  - Read data from the client
  - Send data to the client

3️⃣ `private BufferedReader in;`

- `BufferedReader` reads **text data** from the client.
- It wraps the **InputStream** of the socket for easier reading.

4️⃣ `private PrintWriter out;`

- `PrintWriter` is used to **send messages back to the client**.
- It wraps the **OutputStream** of the socket for easier writing.

5️⃣ Constructor: `public ClientHandler(Socket socket)`

```
public ClientHandler(Socket socket) {
        this.socket = socket;
}
```

- When a new client connects:
  - The server passes its `Socket` to this constructor.
  - The `socket` field is initialized for this client.
- Later, you'll probably **initialize `in` and `out` inside `run()`** to handle communication.

```java
@Override
public void run() {
    try {
        // Initialize input/output streams
        in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        out = new PrintWriter(socket.getOutputStream(), true);

        // Add this client's writer to the set
        synchronized (clientWriters) {
            clientWriters.add(out);
        }

        // Read messages from client and broadcast
        String message;
        while ((message = in.readLine()) != null) {
            System.out.println("Received: " + message);
            broadcastMessage(message, out); // pass the sender
        }

    } catch (IOException e) {
        System.out.println("Client disconnected: " + socket.getInetAddress());
    } finally {
        try { socket.close(); } catch (IOException ignored) {}
        synchronized (clientWriters) {
            clientWriters.remove(out);
        }
    }
}
```

1️⃣ `in = new BufferedReader(new InputStreamReader(socket.getInputStream()));`

- Gets the **input stream** from the client's socket.
- Wraps it in a `BufferedReader` for **easy reading of text lines**.
- This allows the server to **read messages sent by the client**.

2️⃣ `out = new PrintWriter(socket.getOutputStream(), true);`

- Gets the **output stream** of the client's socket.
- Wraps it in a `PrintWriter` for **easy sending of text messages**.
- `true` means **auto-flush**: messages are sent immediately without buffering.

3️⃣ `synchronized (clientWriters) { clientWriters.add(out); }`

- Adds this client's `PrintWriter` to the **set of all clients**.
- `synchronized` ensures **thread safety**:
  - Multiple clients might connect/disconnect at the same time
  - This prevents **race conditions**.

4️⃣ The main loop:

```
String message;
while ((message = in.readLine()) != null) {
        System.out.println("Received: " + message);
        broadcastMessage(message, out);
}
```

- `in.readLine()` waits for the client to send a message.
- If the client disconnects, it returns `null` and **exits the loop**.
- `System.out.println("Received: " + message)` prints messages on the **server console**.
- `broadcastMessage(message, out)` sends the message to **all other clients**.

5️⃣ Handling exceptions:

```
} catch (IOException e) {
        System.out.println("Client disconnected: " +
socket.getInetAddress());
}
```

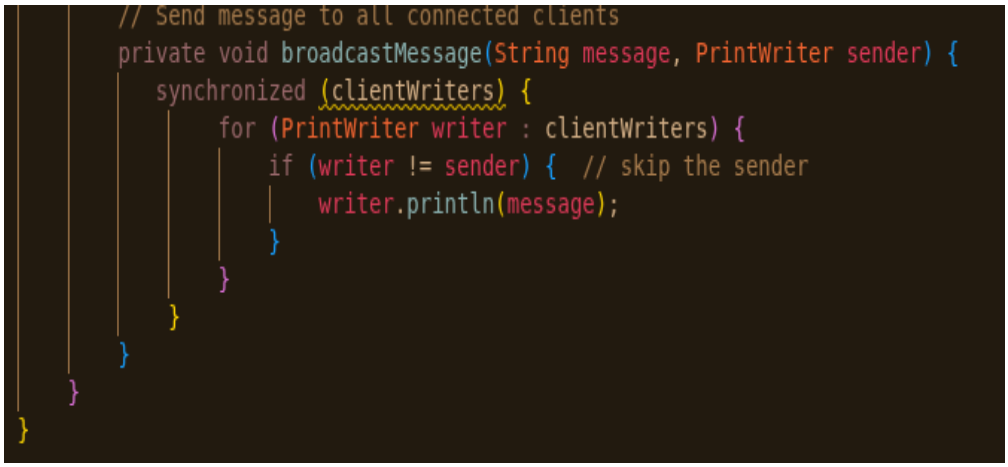
- If a client disconnects or there is a network error, the **catch block executes**.
- Prints which client disconnected.

6️⃣ Cleanup in `finally`:

```
finally {
        try { socket.close(); } catch (IOException ignored) {}
        synchronized (clientWriters) {
        clientWriters.remove(out);
```

```
        }
    }
```

- Ensures the client's socket is **always closed**.
- Removes the client from `clientWriters` so we don't try to send messages to a **disconnected client**.



1️⃣ Method declaration **`private void broadcastMessage(String message, PrintWriter sender)`**

- `message` → The text that one client sent.
- `sender` → The `PrintWriter` of the client who sent the message.
- Purpose: **send the message to all other connected clients except the sender**.

2️⃣ `synchronized (clientWriters)`

- Ensures **thread safety** because multiple clients could be sending messages at the same time.
- Prevents errors when adding/removing writers while iterating.

3️⃣ The loop

```
for (PrintWriter writer : clientWriters) {
        if (writer != sender) {  // skip the sender
writer.println(message);
        }
}
```

- Iterates through **all connected clients**.

- `if (writer != sender)` → Makes sure the client who sent the message **does not receive it again**.
- `writer.println(message)` → Sends the message to that client.

for Class SocketGUI.Java

```
1  package socket;
2
3  import javax.swing.*;
4  import java.awt.*;
5  import java.io.*;
6  import java.net.Socket;
```

- `socket package` → `code organization`
- `javax.swing.*` → GUI components
- `java.awt.*` → GUI layouts, colors, fonts
- `java.io.*` → Reading and sending data
- `java.net.Socket` → Networking

```
public class SocketGUI extends javax.swing.JFrame {

    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;

    private JPanel chatContainer;

    public SocketGUI() {
        initComponents();
        initChatContainer();
        connectToServer();
        MessageTextField.addActionListener(e -> SendButtonActionPerformed(null));
    }

    // Initialize the panel that holds chat bubbles inside your existing scroll pane
    private void initChatContainer() {
        chatContainer = new JPanel();
        chatContainer.setLayout(new BoxLayout(chatContainer, BoxLayout.Y_AXIS));
        chatContainer.setBackground(new Color(153, 255, 204));

        // top padding
        chatContainer.add(Box.createVerticalStrut(10));

        jScrollPane1.setViewportView(chatContainer);
}
```

- This code **sets up chat window**.
- Prepares a **scrollable chat panel** (`chatContainer`) for messages.
- Connects to the server and sets **Enter key behavior** for sending messages.

```
private void addBubble(String message, boolean isSender) {
    ChatBubble bubble = new ChatBubble(message, isSender);
    chatContainer.add(bubble);
    chatContainer.add(Box.createVerticalStrut(5)); // spacing

    chatContainer.revalidate();
    chatContainer.repaint();

    // auto-scroll
    SwingUtilities.invokeLater(() -> {
        JScrollBar bar = jScrollPane1.getVerticalScrollBar();
        bar.setValue(bar.getMaximum());
    });
}

private void SendButtonActionPerformed(java.awt.event.ActionEvent evt) {
    String message = MessageTextField.getText().trim();
    if (message.isEmpty()) {
        return;
    }

    addBubble(message, true);

    if (out != null) {
        out.println(message);
    } else {
        addBubble("❌ Not connected to server", false);
    }

    MessageTextField.setText("");
}
```

1️⃣ `private void addBubble(String message, boolean isSender)`

This method **adds a chat bubble to the GUI**.

```
ChatBubble bubble = new ChatBubble(message, isSender);
chatContainer.add(bubble);
chatContainer.add(Box.createVerticalStrut(5));
```

- ☐ `ChatBubble` → A custom panel for a single message (your chat bubble).
- ☐ `isSender` → If `true`, bubble is aligned for **you**; if `false`, for **other users**.
- ☐ `Box.createVerticalStrut(5)` → Adds **5px spacing** between bubbles.

```
chatContainer.revalidate();
chatContainer.repaint();
```

- ☐ Updates the panel so the **new bubble appears immediately**.

```
SwingUtilities.invokeLater(() -> {
        JScrollBar bar = jScrollPane1.getVerticalScrollBar();
        bar.setValue(bar.getMaximum());
});
```

- ☐ Automatically **scrolls to the bottom** whenever a new message is added.

- ☐ Ensures the **latest message is visible**.

2️⃣ `private void SendButtonActionPerformed(...)`

This method is called when the **Send button is clicked** or **Enter key is pressed**:

```
String message = MessageTextField.getText().trim();
if (message.isEmpty()) return;
```

- ☐ Reads the text from the input field.
- ☐ Ignores if the message is **empty or just spaces**.

```
addBubble(message, true);
```

- ☐ Adds your **own message** to the chat GUI.

```
if (out != null) {
            out.println(message);
} else {
 addBubble("❌ Not connected to server", false);
}
```

- ☐ Sends the message to the **server** using the `PrintWriter out`.
- ☐ If `out` is `null` (not connected), shows an **error bubble**.

```
MessageTextField.setText("");
```

- ☐ Clears the input field after sending.

```
private void connectToServer() {
    new Thread(() -> {
        try {
            socket = new Socket("localhost", 5000);
            in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream(), true);

            addBubble("✅ Connected to server", false);
            receiveMessages();
        } catch (IOException e) {
            addBubble("❌ Server not available", false);
        }
    }).start();
}

private void receiveMessages() {
    try {
        String message;
        while ((message = in.readLine()) != null) {
            addBubble(message, false);
        }
    } catch (IOException e) {
        addBubble("🔌 Disconnected from server", false);
    }
}
```

1. `connectToServer()` → Tries to connect to the server in a **separate thread**.

2. If connected → shows **connected bubble** and calls `receiveMessages()`.
3. `receiveMessages()` → Continuously **listens for incoming messages** and adds them as bubbles.
4. Sending messages → Handled by `SendButtonActionPerformed()`, which uses `out.println(message)` to send messages to the server.
5. Chat bubbles → Handled by `addBubble()` for both sent and received messages.

```
class ChatBubble extends JPanel {

    public ChatBubble(String text, boolean isSender) {
        setLayout(new BorderLayout());
        setOpaque(false);

        JLabel label = new JLabel("<html><p style='width:200px'>" + text + "</p></html>");

        label.setOpaque(true); // ⭐ THIS FIXES COLOR
        label.setForeground(isSender ? Color.WHITE : Color.BLACK);
        label.setBackground(isSender ? new Color(0, 153, 255) : Color.WHITE);

        label.setBorder(
            BorderFactory.createCompoundBorder(
                new RoundedBorder(15),
                BorderFactory.createEmptyBorder(10, 15, 10, 15)
            )
        );

        if (isSender) {
            add(label, BorderLayout.EAST);
        } else {
            add(label, BorderLayout.WEST);
        }
    }
}
```

- ☐ Each `ChatBubble` is a **panel containing a JLabel**.
- ☐ Rounded corners + padding + color + alignment → **modern chat look**.
- ☐ Works with `addBubble()` in your main GUI to **display messages dynamically**.

```java
@SuppressWarnings("unchecked")
private void initComponents() {

    jPanel1 = new javax.swing.JPanel();
    jPanel2 = new javax.swing.JPanel();
    TextLabel = new javax.swing.JLabel();
    jPanel3 = new javax.swing.JPanel();
    SendButton = new javax.swing.JButton();
    MessageTextField = new javax.swing.JTextField();
    jScrollPane1 = new javax.swing.JScrollPane();

    setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
    setTitle("Chat App");

    jPanel1.setBackground(new java.awt.Color(153, 255, 204));
    jPanel1.setBorder(new javax.swing.border.SoftBevelBorder(javax.swing.border.BevelBorder.RAISED));

    jPanel2.setBackground(new java.awt.Color(153, 255, 204));

    TextLabel.setFont(new java.awt.Font("DejaVu Serif", 1, 18)); // NOI18N
    TextLabel.setText("Chat App Using Socket Programming");

    javax.swing.GroupLayout jPanel2Layout = new javax.swing.GroupLayout(jPanel2);
    jPanel2.setLayout(jPanel2Layout);
    jPanel2Layout.setHorizontalGroup(
        jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(jPanel2Layout.createSequentialGroup()
                .addContainerGap()
                .addComponent(TextLabel, javax.swing.GroupLayout.DEFAULT_SIZE, 434, Short.MAX_VALUE)
                .addContainerGap())
    );
```

1️⃣ `initComponents()`

- ☐ This method **creates and arranges all GUI components**.
- ☐ Typically auto-generated by **NetBeans or another GUI designer**.
- ☐ Not need to manually change most of it unless customizing colors, fonts, or layouts.

2️⃣ Panels

```
jPanel1 = new JPanel(); // main container
jPanel2 = new JPanel(); // top header panel
jPanel3 = new JPanel(); // bottom input panel
```

- ☐ `jPanel1` → Contains **all other panels** and the scrollable chat area.
- ☐ `jPanel2` → Top panel with **title label**.
- ☐ `jPanel3` → Bottom panel with **message input and Send button**.

3️⃣ Labels and Buttons

```
TextLabel = new JLabel("Chat App Using Socket Programming");
SendButton = new JButton("Send");
MessageTextField = new JTextField();
```

- ☐ `TextLabel` → Chat app title in the top panel.
- ☐ `SendButton` → Clicking it **sends the message**.
- ☐ `MessageTextField` → Where the user types a message.

4️⃣ Scroll Pane

```
jScrollPane1 = new JScrollPane();
jScrollPane1.setHorizontalScrollBarPolicy(HORIZONTAL_SCROLLBAR_NEVER);
```

- ☐ The scroll pane **contains the chatContainer** panel.
- ☐ Ensures **vertical scrolling** when messages exceed visible area.
- ☐ Horizontal scroll is disabled; text wraps in chat bubbles.

5️⃣ Layouts

- ☐ `GroupLayout` is used to arrange components **precisely** in each panel.
- ☐ `jPanel1Layout` → Arranges: **header (top)**, **scroll chat area (center)**, **input panel (bottom)**.
- ☐ `jPanel2Layout` → Centers the title label.
- ☐ `jPanel3Layout` → Places text field and Send button horizontally.

6️⃣ Colors, Fonts, and Styles

```
jPanel1.setBackground(new Color(153, 255, 204)); // light green
MessageTextField.setBackground(new Color(255, 255, 153)); // light yellow
SendButton.setBackground(new Color(153, 255, 204));
TextLabel.setFont(new Font("DejaVu Serif", Font.BOLD, 18));
```

- ☐ Panels and components have **custom colors and fonts** for a nicer look.
- ☐ Send button uses a **raised bevel border** to make it look clickable.

7️⃣ Pack and Display

```
pack();
```

- ☐ `pack()` → Adjusts window size **automatically** based on component sizes.

- Ensures the window fits all components neatly.

```
    Run main | Debug main
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new SocketGUI().setVisible(true));
    }

    private javax.swing.JPanel jPanel1;
    private javax.swing.JPanel jPanel2;
    private javax.swing.JPanel jPanel3;
    private javax.swing.JLabel TextLabel;
    private javax.swing.JButton SendButton;
    private javax.swing.JTextField MessageTextField;
    private javax.swing.JScrollPane jScrollPane1;
}
```

```
public static void main(String[] args)
        SwingUtilities.invokeLater(() -> new SocketGUI().setVisible(true));
```

- This is the **starting point of the program**.
- `SwingUtilities.invokeLater(...)` → Ensures that **all GUI components are created on the Event Dispatch Thread (EDT)**.
  - In Swing, **all GUI updates must happen on the EDT** to avoid weird bugs.
- `new SocketGUI().setVisible(true)`
  - Creates a **new instance of the chat window**.
  - Makes the window **visible**.

for Class RoundedBorder.Java

```
public class RoundedBorder extends AbstractBorder {

    private final int radius;

    public RoundedBorder(int radius) {
        this.radius = radius;
    }

    @Override
    public void paintBorder(
            Component c,
            Graphics g,
            int x,
            int y,
            int width,
            int height
    ) {
        Graphics2D g2 = (Graphics2D) g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                            RenderingHints.VALUE_ANTIALIAS_ON);

        g2.setColor(c.getForeground());
        g2.drawRoundRect(
                x, y,
                width - 1,
                height - 1,
                radius,
                radius
        );
    }
```

1️⃣ `public class RoundedBorder extends AbstractBorder`

- ☐ Extends **AbstractBorder**, which allows to **customize borders** for Swing components.
- ☐ This will be used in `ChatBubble` like:

```
label.setBorder(new RoundedBorder(15));
```

2️⃣ Field and constructor

```
private final int radius;
public RoundedBorder(int radius) {
        .radius = radius;
}
```

- ☐ `radius` → Controls **how rounded the corners are**.
- ☐ `final` → Value cannot be changed after creation.

3️⃣ `paintBorder(...)` method

```
Graphics2D g2 = (Graphics2D) g;
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                    RenderingHints.VALUE_ANTIALIAS_ON);
```

- ☐ Converts `Graphics` to `Graphics2D` for **better drawing features**.
- ☐ Enables **anti-aliasing** → smooth edges instead of jagged.

```
g2.setColor(c.getForeground());
g2.drawRoundRect(x, y, width - 1, height - 1, radius, radius);
```

- ☐ `c.getForeground()` → Uses the component's **foreground color** for the border.
- ☐ `drawRoundRect(...)` → Draws a rectangle with **rounded corners**:
  - ☐ `x, y` → top-left corner
  - ☐ `width-1, height-1` → size of the rectangle
  - ☐ `radius` → corner curvature

```
    @Override
    public Insets getBorderInsets(Component c, Insets insets) {
        insets.left = insets.right = 15;
        insets.top = insets.bottom = 10;
        return insets;
    }
}
```

1️⃣ Method signature

```
@Override
public Insets getBorderInsets(Component c, Insets insets)
```

- ☐ `Insets` defines **space between the border and the component's content** (padding).
- ☐ `c` → The component this border is applied to (like `JLabel`).
- ☐ `insets` → The current insets object that can be modified.

2️⃣ Setting the padding

```
insets.left = insets.right = 15;
insets.top = insets.bottom = 10;
```

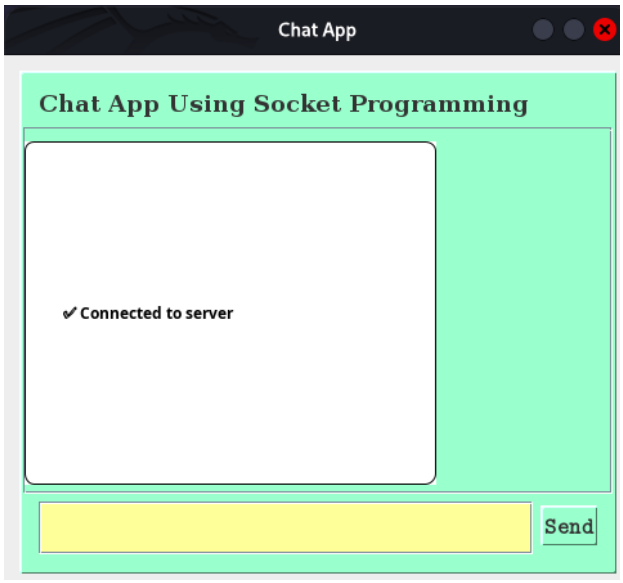
- ☐ This **adds extra space inside the border**:
  - ☐ Left & Right → 15 pixels
  - ☐ Top & Bottom → 10 pixels
- ☐ Ensures **texts inside chat bubbles doesn't touch the edges**.

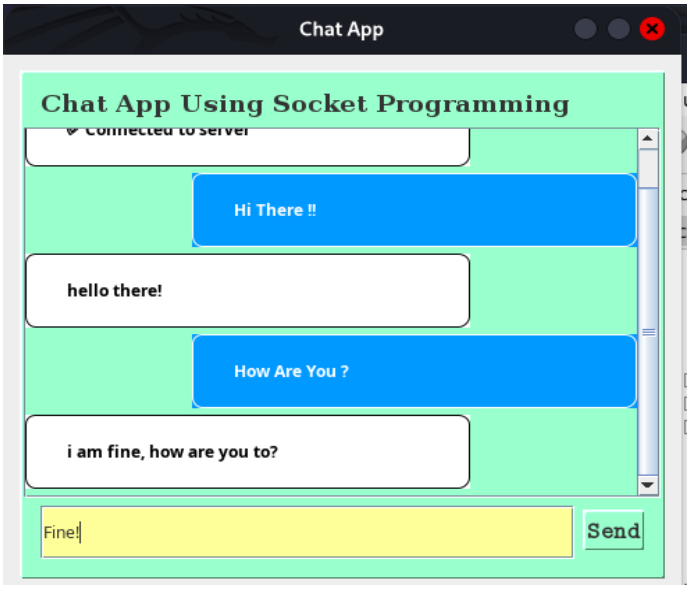
3️⃣ Returning the insets

```
return insets;
```

- ☐ Returns the **modified padding** so Swing knows how much space to leave inside the border.

6. Sample Program Out Put





7. How to Run the Program

- Compile ChatServer.java and SocketGUI.java using javac
  - Find the code that is same with this on SsocketUI.java file (line 70)

```
private void connectToServer() {
        new Thread(() -> {
            try {
            socket = new Socket("localhost", 5000);
            in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream(), true);
            addBubble("✅ Connected to server", false);
            receiveMessages();
            } catch (IOException e) {
            addBubble("❌ Server not available", false);
            }
        }).start();
    }
```

  - Convert the localhost to Server's Ip Address on the client side (192.168.107.75)
- Run ChatServer.java first then
- Run SocketGUI.java

8. Conclusion

This documentation presented a Java multi-client chat application using socket programming. The system successfully supports multiple clients, real-time communication, and a GUI interface.