JAVA CASE STUDY GROUP PROJECT

[AQIB JAWED]

[2023006524]

#8

***Case Study**: **Chat Application (Multithreading, Networking):-**

#Real-Time Chat Application Using Java Networking & Multithreading

1. Introduction

Communication is an essential part of modern applications, and chat applications enable real-time message exchange between users. This case study presents a Client-Server Chat Application built using Java Sockets and Multithreading, ensuring seamless two-way communication.

2. Objectives

- **Develop a real-time chat system where multiple users can communicate.**
- **Implement Client-Server communication using Sockets.**
- **Use Multithreading to handle multiple clients simultaneously.**
- **Ensure efficient and reliable message exchange.**

3. Technologies & Concepts Used

-Networking: Socket, ServerSocket for data transmission.

-Multithreading: Thread for handling multiple clients.

-Java I/O: BufferedReader, PrintWriter for message exchange.

-Exception Handling: Ensuring robustness against errors.

4. <u>System Design</u>

My chat system follows a Client-Server Architecture, where:

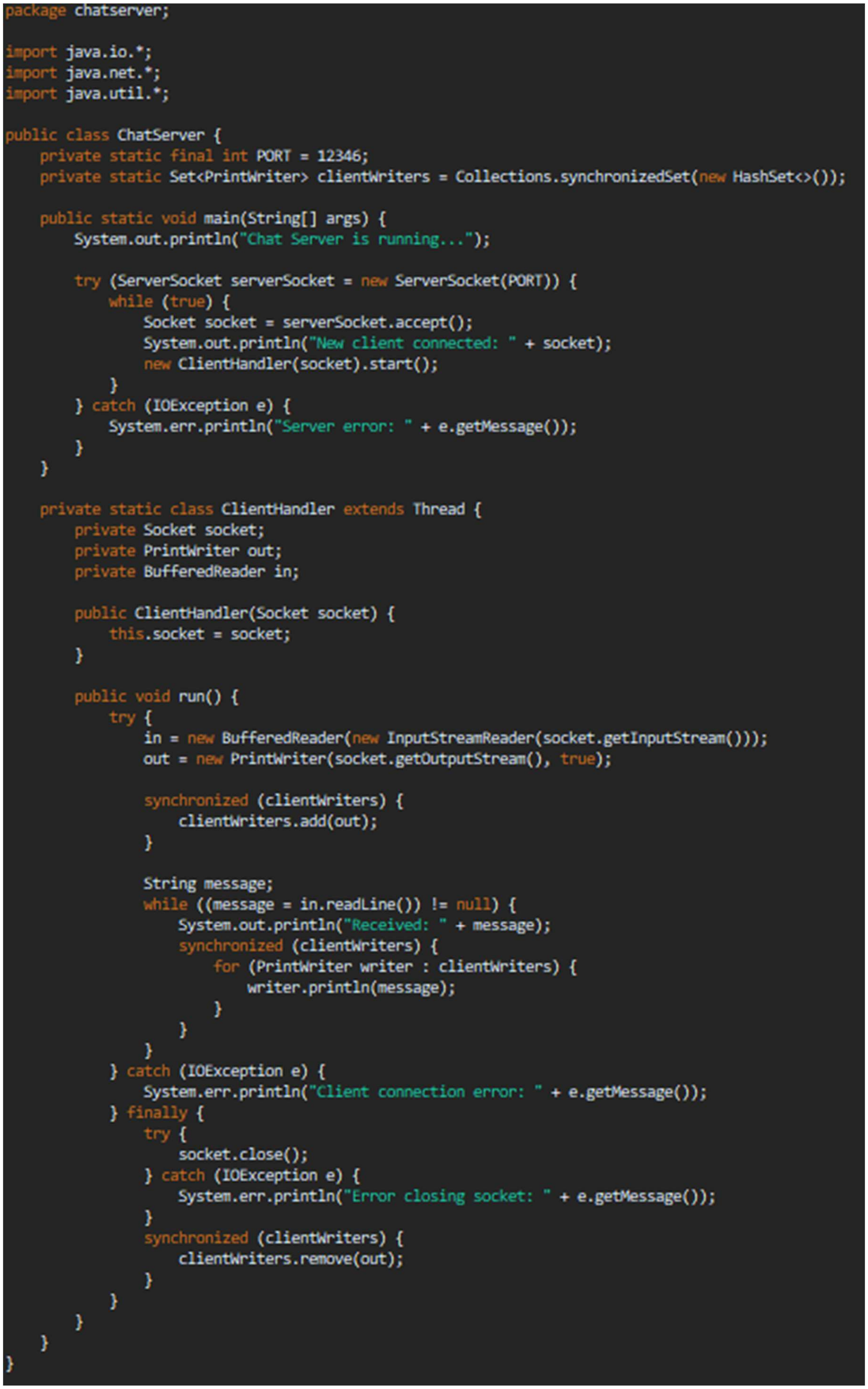
- **The server listens for connections from multiple clients.**
- **Each client connects to the server and can send/receive messages.**
- **Multithreading is used to handle multiple clients concurrently.**

Components:

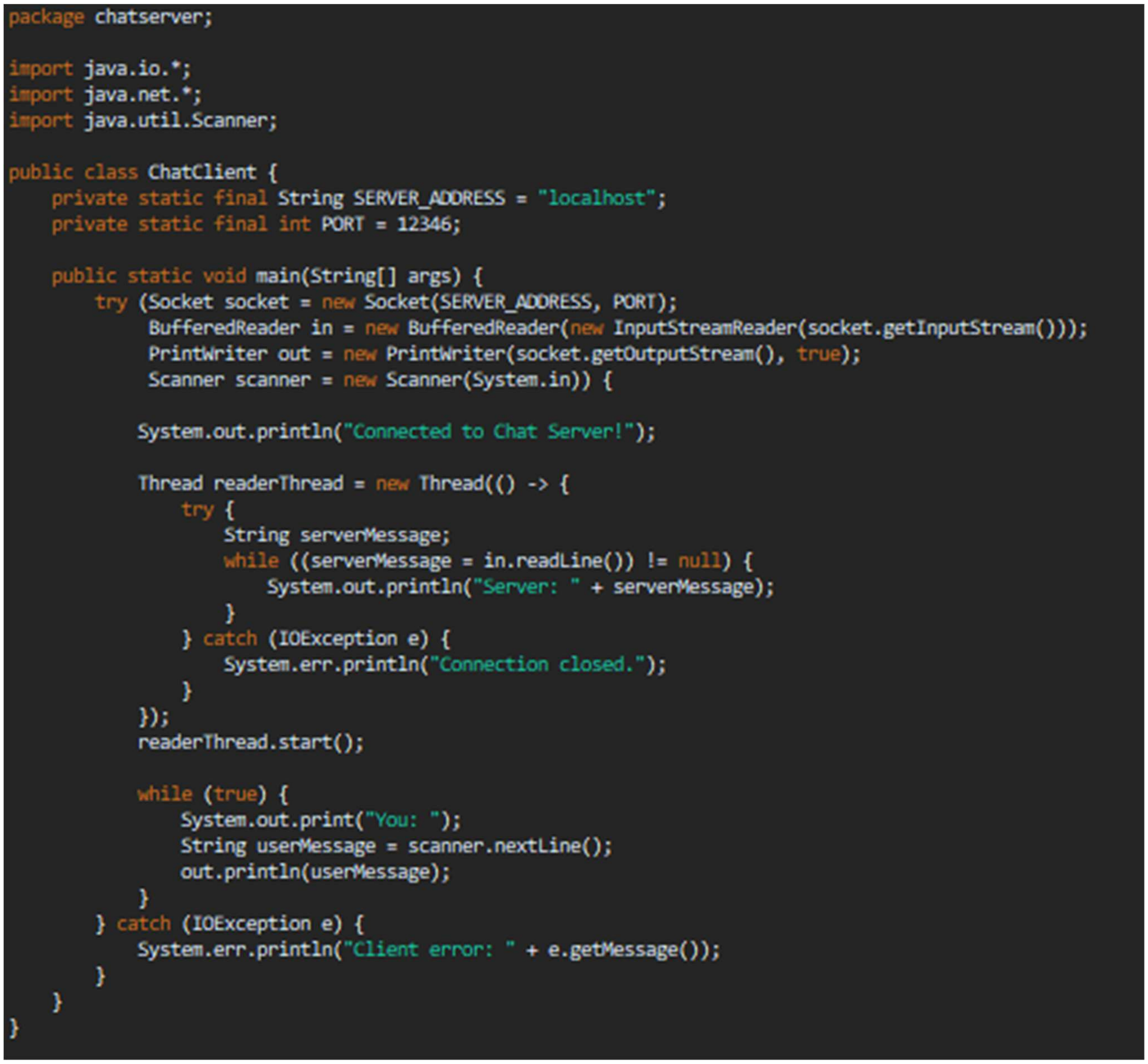
- **Server: Handles multiple client connections.**
- **Client: Connects to the server and allows user input.**

5. <u>Implementation</u>

<u>Step 1</u>: Server Code

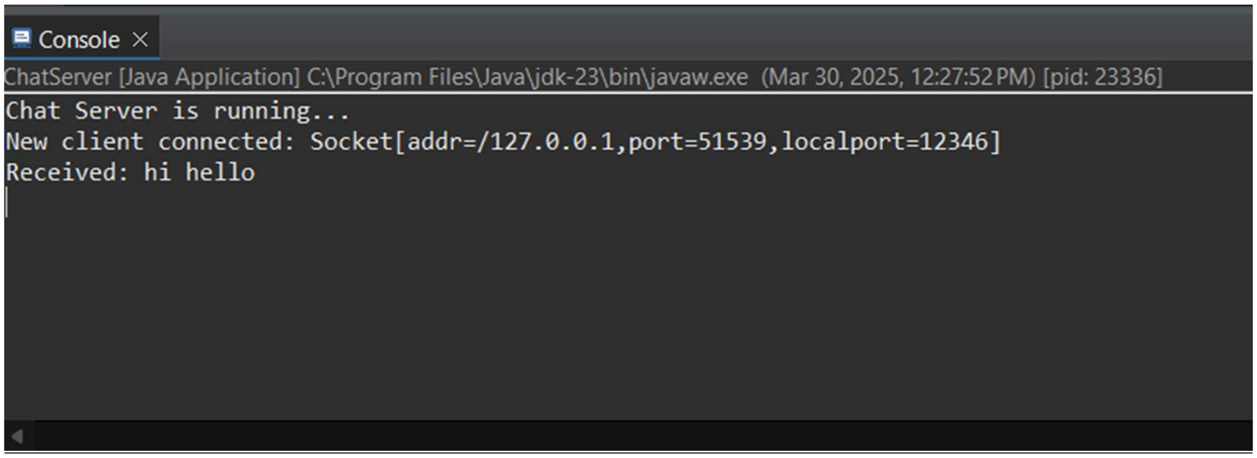

```
package chatserver;

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

public class ChatServer {
    private static final int PORT = 12346;
    private static Set<PrintWriter> clientWriters = Collections.synchronizedSet(new HashSet<>());

    public static void main(String[] args) {
        System.out.println("Chat Server is running...");

        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            while (true) {
                Socket socket = serverSocket.accept();
                System.out.println("New client connected: " + socket);
                new ClientHandler(socket).start();
            }
        } catch (IOException e) {
            System.err.println("Server error: " + e.getMessage());
        }
    }

    private static class ClientHandler extends Thread {
        private Socket socket;
        private PrintWriter out;
        private BufferedReader in;

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

        public void run() {
            try {
                in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
                out = new PrintWriter(socket.getOutputStream(), true);

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

                String message;
                while ((message = in.readLine()) != null) {
                    System.out.println("Received: " + message);
                    synchronized (clientWriters) {
                        for (PrintWriter writer : clientWriters) {
                            writer.println(message);
                        }
                    }
                }
            } catch (IOException e) {
                System.err.println("Client connection error: " + e.getMessage());
            } finally {
                try {
                    socket.close();
                } catch (IOException e) {
                    System.err.println("Error closing socket: " + e.getMessage());
                }
                synchronized (clientWriters) {
                    clientWriters.remove(out);
                }
            }
        }
    }
}
```

Step 2: Client Code



```java
package chatserver;

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

public class ChatClient {
    private static final String SERVER_ADDRESS = "localhost";
    private static final int PORT = 12346;

    public static void main(String[] args) {
        try (Socket socket = new Socket(SERVER_ADDRESS, PORT);
             BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
             PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
             Scanner scanner = new Scanner(System.in)) {

            System.out.println("Connected to Chat Server!");

            Thread readerThread = new Thread(() -> {
                try {
                    String serverMessage;
                    while ((serverMessage = in.readLine()) != null) {
                        System.out.println("Server: " + serverMessage);
                    }
                } catch (IOException e) {
                    System.err.println("Connection closed.");
                }
            });
            readerThread.start();

            while (true) {
                System.out.print("You: ");
                String userMessage = scanner.nextLine();
                out.println(userMessage);
            }
        } catch (IOException e) {
            System.err.println("Client error: " + e.getMessage());
        }
    }
}
```

Output:



```
ChatServer [Java Application] C:\Program Files\Java\jdk-23\bin\javaw.exe  (Mar 30, 2025, 12:27:52 PM) [pid: 23336]
Chat Server is running...
New client connected: Socket[addr=/127.0.0.1,port=51539,localport=12346]
Received: hi hello
```

6. <u>Working of the Application</u>

1. **Start the server: Run ChatServer.java, which waits for client connections.**
2. **Start clients: Run multiple instances of ChatClient.java to connect to the server.**
3. **Send messages: Clients can send messages, and the server will broadcast them to all connected clients.**
4. **Real-time Communication: Messages are sent and received instantly using threads.**

7. <u>Exception Handling</u>

- **Handling Client Disconnections: When a client disconnects, the server removes it from the active list.**
- **Input/Output Errors: Try-catch blocks ensure unexpected crashes do not affect other users.**
- **Port Availability: The server ensures the port is available before starting.**

8. <u>Advantages of the Approach</u>

-Real-time communication with instant message exchange.
-Efficient use of Multithreading to handle multiple clients.
-Networking using Sockets ensures reliable data transfer.
-Scalability: Can be extended with GUI, encryption, etc.

9. Conclusion

This case study successfully demonstrates a real-time chat application using Java Networking & Multithreading. By leveraging Sockets, Threads, and Exception Handling, we achieved seamless client-server communication. This project can be extended further by integrating GUI, databases, or security features.

10. References

1. **Java Networking Documentation – Oracle Docs**
2. **Java Multithreading – GeeksforGeeks**
3. **Socket Programming – Baeldung**
