НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

КАФЕДРА АВТОМАТИКИ ТА УПРАВЛІННЯ В ТЕХНІЧНИХ СИСТЕМАХ

**Лабораторна робота № 2**

**з предмету «Програмування комп’ютерних та віртуальних мереж» на тему «Створити багатопотокового клієнт-серверного застосунку»**

Виконала:

студентка групи ІТ-з03мп

Козак О. С.

Перевірив:

Галушко Д.О.

**Київ – 2020р.**

**Лабораторна робота № 2**

**Мета:** Створити багатопотоковий клієнт-серверний застосунок: чат-клієнт та чат-сервер.

**Завдання:** Реалізувати наступний клієнт-серверний застосунок:

Сервер:

- Очікує запити на з’єднання від клієнтів (номер порту задається під час запуску застосунку як параметр командного рядка);

- Встановлює та підтримує з’єднання з клієнтами, розсилає інформацію усім клієнтам про те, що до чату приєднався новий клієнт;

- У разі надходження повідомлення від одного з клієнтів надсилає це повідомлення усім клієнтам;

- У разі від’єднання одного з клієнтів розсилає про це інформацію решті клієнтів;

- Має бути реалізований як консольний застосунок;

- Сервер має бути реалізований як багатопоточний застосунок «1 клієнт = 1 потік» використовуючи блокуючий ввід-вивід.

Клієнт:

- Клієнт має з’єднуватись з сервером (адреса сервера, номер порту та ім’я користувача задаються як параметри командного рядка);

- Клієнт має бути реалізований як багатопоточний застосунок. Один потік має очікувати введення тексту користувачем і відправляти його на сервер, а інший потік – виводити на екран повідомлення, які клієнт отримує від сервера;

- Клієнт може бути реалізований як консольний або десктопний застосунок.

**Хід роботи:**

Створення багатопотокового клієнт-серверного застосунку ми реалізували використовуючи наслідування класу Thread та реалізації методу run**.**

**1.** Створюємо сервер.

а) Клас Server відповідає за створення сокету сервера. Створюємо сокет сервера, використовуючи клас ServerSocket. У цьому ж класі ми створюємо LinkedList, який буде відповідати за підключення до клієнтів. Номер порту задається як параметр командного рядка.

**Лістинг Server:**

package CNVP\_Lab2\_Server;  
  
import java.io.IOException;  
import java.net.InetAddress;  
import java.net.ServerSocket;  
import java.util.LinkedList;  
import java.util.Scanner;  
  
public class Server {  
 public static LinkedList<ClientListenerThread> *serverList* = new LinkedList<>();  
 private static ServerSocket *server*;  
 public static boolean *isRunning* = true;  
  
 public static ServerSocket getServer() {  
 if (*server* == null) {  
 Scanner scanner = new Scanner(System.*in*);  
 System.*out*.println("enter port");  
 int port = scanner.nextInt();  
 try {  
 *server* = new ServerSocket(port, 10, InetAddress.*getByName*("127.0.0.1")); } catch (IOException exception) {  
 exception.printStackTrace();  
 }  
 }  
 return *server*;  
 }  
}

б) У методі main ми створюємо сокет сервера, встановлюємо з’єднання з клієнтами. Встановивши з’єднання створюється для кожного клієнта потік(thread), за що відповідає клас ClientListenerThread. Потік(thread) клієнта буде приймати запити та надавати відповідь на них. Всі потоки(thread) клієнтів ми додаємо та зберігаємо у Server.serverList. Разом з тим, створюємо другий потік(thread), за який відповідає клас ServerShutdownThread, який буде розривати з’єднання з боку сервера, якщо буде введено в командний рядок запит на завершення сеансу.

**Лістинг Main:**

package CNVP\_Lab2\_Server;  
  
import java.io.IOException;  
import java.net.ServerSocket;  
import java.net.Socket;  
  
  
public class Main {  
  
 public static void main(String[] args) throws IOException, InterruptedException {  
 ServerSocket server = Server.*getServer*();  
 System.*out*.println("Server is running");  
 ServerShutdownThread serverShutdownThread = new ServerShutdownThread();  
 try {  
 serverShutdownThread.start();  
 while (Server.*isRunning*) {  
 Socket socket = server.accept();  
 System.*out*.println("socket is connected");  
 ClientListenerThread clientListenerThread = new ClientListenerThread(socket);  
 Server.*serverList*.add(clientListenerThread);  
 }  
 serverShutdownThread.join();  
 } catch (IOException ioException) {  
 if (!Server.*isRunning*) {  
 System.*out*.println("server closed");  
 } else {  
 throw ioException;  
 }  
 } finally {  
 server.close();  
 }  
 }  
}

в) Розглянемо клас ClientListenerThread, клас який приймає запити від клієнтів.Створюємойого, наслідуючи клас Thread, який потрібен нам для створення потоків (thread) та робимо override методу run. У конструкторі класу створюємо буфери для записування у потік (stream) та зчитування з нього. У методі run сервер приймає дані від клієнтів та розсилає це повідомлення всім клієнтам, окрім того, хто надіслав повідомлення. Метод *shutdown* закриває сокет, якщо від клієнта приходить запит на відключення, а також всі існуючи буфери.

**Лістинг ClientListenerThread**

package CNVP\_Lab2\_Server;  
  
import java.io.\*;  
import java.net.Socket;  
import java.net.SocketException;  
  
public class ClientListenerThread extends Thread {  
 public Socket socket;  
 private BufferedReader reader;  
 private BufferedWriter writer;  
  
  
 public ClientListenerThread(Socket socket) throws IOException {  
 this.socket = socket;  
 reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));  
 writer = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));  
 start();  
 }  
  
 @Override  
 public void run() {  
 System.*out*.println("Listener thread started for " + this.socket.getInetAddress());  
 try {  
 while (Server.*isRunning*) {  
 if (Server.*serverList*.size() > 0) {  
 String message = reader.readLine();  
 System.*out*.println(message);

String operation = JsonParser.*deserializeOperation*(message);  
 for (ClientListenerThread thread : Server.*serverList*) {  
 if (thread != this) {  
 thread.send(message);  
 }  
 }  
 if (operation.equals("Disconnect")) {  
 this.shutdown();  
 break;  
 }  
 } else break;  
 }  
  
 } catch (SocketException exception) {  
 if (!Server.*isRunning*) {  
 System.*out*.println("Listener thread stopped for " + this.socket.getInetAddress());  
 } else {  
 exception.printStackTrace();  
 }  
 } catch (IOException exception) {  
 exception.printStackTrace();  
 }  
 }  
  
 private void send(String message) {  
 try {  
 writer.write(message + "\n");  
 writer.flush();  
 } catch (IOException exception) {  
 System.*out*.println("failed to send message");  
 exception.printStackTrace();  
 }  
 }  
  
 private void shutdown() {  
 try {  
 if (!socket.isClosed()) {  
 socket.close();  
 }  
 reader.close();  
 writer.close();  
 Server.*serverList*.remove(this);  
 } catch (IOException exception) {  
 exception.printStackTrace();  
 }  
 }  
}

г) Оскільки у нас повідомлення представлені у вигляді Json, за десеріалізацію відповідає клас JsonParser.

**Лістинг JsonParser:**

package CNVP\_Lab2\_Server;  
  
import com.fasterxml.jackson.databind.JsonNode;  
import com.fasterxml.jackson.databind.ObjectMapper;  
  
import java.io.IOException;  
  
public class JsonParser {  
 public static String deserializeOperation(String received) throws IOException {  
 ObjectMapper objectMapper = new ObjectMapper();  
 JsonNode json = objectMapper.readTree(received);  
 String operation = json.get("operation").textValue();  
 return operation;  
 }  
}

д) Розглянемо клас ServerShutdownThread, який відповідає за розрив з’єднання між сервером та клієнтами.

**Лістинг ServerShutdownThread:**

package CNVP\_Lab2\_Server;  
  
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStreamReader;  
  
public class ServerShutdownThread extends Thread {  
 public static BufferedReader *clientInput*;  
  
 @Override  
 public void run() {  
 try {  
 String inputMessage;  
 *clientInput* = new BufferedReader(new InputStreamReader(System.*in*));  
 while (true) {  
 System.*out*.println("To stop server enter \"stop\"");  
 inputMessage = *clientInput*.readLine();  
 if (inputMessage.equals("stop")) {  
 Server.*isRunning* = false;  
 System.*out*.println("server is closed");  
 Server.*getServer*().close();  
 for (ClientListenerThread clientListenerThread : Server.*serverList*) {  
 clientListenerThread.socket.close();  
 }  
 break;  
 } else {  
 Server.*isRunning* = true;  
 }  
 }  
 } catch (IOException exception) {  
 exception.printStackTrace();  
 }  
 }  
}

2.Створюємо клієнт.

а) Метод *createSocket* клієнт створює сокет, в якому задає IP-адресу серверу. Створюється BufferedReader та BufferedWriter, а також буфер для зчитування повідомлення клієнта з командного рядку.

**Лістинг Client:**

package CNVP\_Lab2\_CLient;  
  
import java.io.\*;  
import java.net.InetAddress;  
import java.net.Socket;  
import java.util.Scanner;  
  
public class Client {  
  
 public static BufferedReader *reader*;  
 public static BufferedWriter *writer*;  
 public static BufferedReader *clientInput*;  
  
 public static Socket *clientSocket*;  
 public static boolean *isShutdownRequested*;  
  
 public static void run() {  
 try {  
 *createSocket*();  
 *reader* = new BufferedReader(new InputStreamReader(*clientSocket*.getInputStream()));  
 *writer* = new BufferedWriter(new OutputStreamWriter(*clientSocket*.getOutputStream()));  
 *clientInput* = new BufferedReader(new InputStreamReader(System.*in*));  
  
 UserNameHandler.*LoginToServer*();  
  
 ReadMessageThread readMessageThread = new ReadMessageThread();  
 readMessageThread.start();  
 WriteMessageThread writeMessageThread = new WriteMessageThread();  
 writeMessageThread.start();  
 try {  
 writeMessageThread.join();  
 readMessageThread.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 } catch (IOException ex) {  
 ex.printStackTrace();  
 }  
 *shutdown*();  
 }  
  
 private static void createSocket() throws IOException {  
 if (*clientSocket* == null) {  
 InetAddress ipAddressOfServer = InetAddressInput.*getInetAddress*();  
 Scanner scanner = new Scanner(System.*in*);  
 System.*out*.println("enter port");  
 int port = scanner.nextInt();  
 *clientSocket* = new Socket(ipAddressOfServer, port);  
 if (*clientSocket*.isConnected()) {  
 System.*out*.println("Connected to server" + "\n");  
 }  
 }  
 }  
  
 private static void shutdown() {  
 try {  
 if (!*clientSocket*.isClosed()) {  
 *clientSocket*.close();  
 }  
 Client.*reader*.close();  
 Client.*writer*.close();  
 System.*out*.println("client is closed");  
  
 } catch (IOException exception) {  
 exception.printStackTrace();  
 }  
 }  
}

б) Клієнт надсилає серверу три запити: запит на підключення, запит на розсилання повідомлення та запит на розрив з’єднання. Відповідно до цього ми маємо класи для серіалізації: UserConnectionServerRequest, UserDisconnectionServerRequest та MessageServerRequest. Оскільки у нас серіалізація та десеріалізація використовує різні об’єкти, ми додатково обгортаємо їх у обгортку, використовуючи клас ServerRequestWrapper. Клас UserNameHandler зчитує username та надсилає серверу username.

**Лістинг UsernameHandler:**

package CNVP\_Lab2\_CLient;  
  
import com.fasterxml.jackson.core.JsonProcessingException;  
  
import java.io.IOException;  
  
public class UserNameHandler {  
  
 private static String getSerializedName(String name) throws JsonProcessingException {  
 ServerRequestWrapper wrapper = new ServerRequestWrapper();  
 UserConnectionServerRequest userConnectionServerRequest = new UserConnectionServerRequest();  
 userConnectionServerRequest.setUserName(name);  
 userConnectionServerRequest.setDateTime(DateTimeHelper.*getFormattedDateTime*());  
 wrapper.setOperation("UserConnection");  
 wrapper.setData(userConnectionServerRequest);  
 return JsonParser.*serialize*(wrapper);  
 }  
  
 public static void LoginToServer() throws IOException {  
 String userName = UserName.*inputYourName*();  
 String json = UserNameHandler.*getSerializedName*(userName) + "\n";  
 Client.*writer*.write(json);  
 Client.*writer*.flush();  
 }  
}

в) Після надсилання імені користувача, створюються два потоки (thread): один потік надсилає повідомлення серверу (та іншим користувачам), а інший отримує повідомлення. Потоки працюють паралельно.

г) Розглянемо потік (thread) WriteMessageThread для надсилання повідомлень. Клас WriteMessageThread наслідує Thread.

1. Клієнт зчитує повідомлення з командного рядку та, викликаючи метод *sendMessage* надсилає повідомлення на сервер.

**Лістинг WriteMessageThread:**

package CNVP\_Lab2\_CLient;  
  
import java.io.\*;  
  
public class WriteMessageThread extends Thread {  
 @Override  
 public void run() {  
 try {  
 while (true) {  
 String inputMessage = readLine();  
 if (inputMessage.equals("stop")) {  
 SendDataService.*sendMessage*(inputMessage);  
 Client.*isShutdownRequested* = true;  
 break;  
 } else {  
 SendDataService.*sendMessage*(inputMessage);  
 }  
 }  
 } catch (InterruptedIOException exception) {  
 System.*out*.println("User input listener thread stopped");  
 } catch (IOException exception) {  
 if (!Client.*clientSocket*.isConnected()) {  
 System.*out*.println("server closed connection");  
 } else {  
 exception.printStackTrace();  
 }  
 } catch (InterruptedException exception) {  
 exception.printStackTrace();  
 }  
 }  
  
 public String readLine() throws IOException, InterruptedException {  
 String s = "";  
 while (true) {  
 if (Client.*isShutdownRequested*) {  
 throw new InterruptedIOException("IO interrupted due to shutdown request");  
 }  
 if (Client.*clientInput*.ready()) {  
 int c = Client.*clientInput*.read();  
 if (c == -1) throw new IOException("Unexpected end of stream");  
 if ((char) c == '\n') break;  
 s += (char) c;  
 } else {  
 *sleep*(100);  
 }  
 }  
 return s;  
 }  
}

1. Метод *sendMessage* класу SendDataService надсилає серверу повідомлення разом з типом операції.

**Лістинг SendDataService:**

package CNVP\_Lab2\_CLient;  
  
import java.io.IOException;  
  
public class SendDataService {  
 public static void sendMessage(String message) throws IOException {  
 ServerRequestWrapper wrapper = new ServerRequestWrapper();  
  
 if (message.equals("stop")) {  
 UserDisconnectionServerRequest disconnection = new UserDisconnectionServerRequest();  
 disconnection.setUserName(UserName.*name*);  
 disconnection.setDateTime(DateTimeHelper.*getFormattedDateTime*());  
 wrapper.setOperation("Disconnect");  
 wrapper.setData(disconnection);  
 String json = JsonParser.*serialize*(wrapper);  
 *send*(json);  
 } else {  
 MessageServerRequest messageServerRequest = new MessageServerRequest();  
 messageServerRequest.setData(message);  
 messageServerRequest.setUserName(UserName.*name*);  
 messageServerRequest.setDateTime(DateTimeHelper.*getFormattedDateTime*());  
 wrapper.setOperation("Message");  
 wrapper.setData(messageServerRequest);  
 String json = JsonParser.*serialize*(wrapper);  
 *send*(json);  
 }  
 }  
  
 private static void send(String data) throws IOException {  
 Client.*writer*.write(data);  
 Client.*writer*.flush();  
 }  
}

1. Клас JsonParser відповідає за серіалізацію та десеріалізацію запитів.

**Лістинг JsonParser:**

package CNVP\_Lab2\_CLient;  
  
import com.fasterxml.jackson.core.JsonProcessingException;  
import com.fasterxml.jackson.databind.ObjectMapper;  
  
import java.io.IOException;  
  
public class JsonParser {  
  
 public static String serialize(ServerRequestWrapper serverRequestWrapper) throws JsonProcessingException {  
 ObjectMapper objectMapper = new ObjectMapper();  
 return objectMapper.writeValueAsString(serverRequestWrapper);  
 }  
  
 public static UserConnectionServerRequest deserializeUserConnectionRequest(String receivedData) throws IOException {  
 ObjectMapper objectMapper = new ObjectMapper();  
 UserConnectionServerRequest userConnectionServerRequest = objectMapper.readValue(receivedData, UserConnectionServerRequest.class);  
 return userConnectionServerRequest;  
 }  
  
 public static MessageServerRequest deserializeMessageServerRequest(String receivedData) throws IOException {  
 ObjectMapper objectMapper = new ObjectMapper();  
 MessageServerRequest messageServerRequest = objectMapper.readValue(receivedData, MessageServerRequest.class);  
 return messageServerRequest;  
 }  
  
 public static UserDisconnectionServerRequest deserializeDisconnectionRequest(String receivedData) throws IOException {  
 ObjectMapper objectMapper = new ObjectMapper();  
 UserDisconnectionServerRequest userDisconnectionServerRequest = objectMapper.readValue(receivedData, UserDisconnectionServerRequest.class);  
 return userDisconnectionServerRequest;  
 }  
}

1. Оскільки запити потрібно обгорнути у обгортку, ми створюємо клас OperationDispatch, який займається диспетчеризацією операцій (запитів) клієнта.

**Лістинг OperationDispatch:**

package CNVP\_Lab2\_CLient;  
  
import com.fasterxml.jackson.databind.JsonNode;  
import com.fasterxml.jackson.databind.ObjectMapper;  
  
import java.io.IOException;  
  
public class OperationDispatch {  
 public static void dispatch(String receivedData) throws IOException {  
 ObjectMapper objectMapper = new ObjectMapper();  
 JsonNode json = objectMapper.readTree(receivedData);  
 String operation = json.get("operation").textValue();  
 String data = json.get("data").toString();  
 if (operation.equals("UserConnection")) {  
 *UserConnectedHandler*(data);  
 } else if (operation.equals("Message")) {  
 *MessageHandler*(data);  
 } else if (operation.equals("Disconnect")) {  
 *DisconnectionRequest*(data);  
 } else {  
 throw new UnsupportedOperationException("operation " + operation + " is not supported");  
 }  
 }  
  
 public static void UserConnectedHandler(String receivedData) throws IOException {  
 UserConnectionServerRequest userConnectionServerRequest = JsonParser.*deserializeUserConnectionRequest*(receivedData);  
 System.*out*.println(userConnectionServerRequest.toString());  
 }  
  
 public static void MessageHandler(String receivedData) throws IOException {  
 MessageServerRequest messageServerRequest = JsonParser.*deserializeMessageServerRequest*(receivedData);  
 System.*out*.println(messageServerRequest.toString());  
 }  
  
 public static void DisconnectionRequest(String receivedData) throws IOException {  
 UserDisconnectionServerRequest userDisconnectionServerRequest = JsonParser.*deserializeDisconnectionRequest*(receivedData);  
 System.*out*.println(userDisconnectionServerRequest.toString());  
 }  
}

д) Розглянемо ReadMessageThread. Використовуючи метод *readLine* зчитуємо повідомлення від серверу. Метод OperationDispatch.*dispatch* виконує десеріалізаціюта передає повідомлення у командний рядок клієнта.

Лістинг ReadMessageThread:

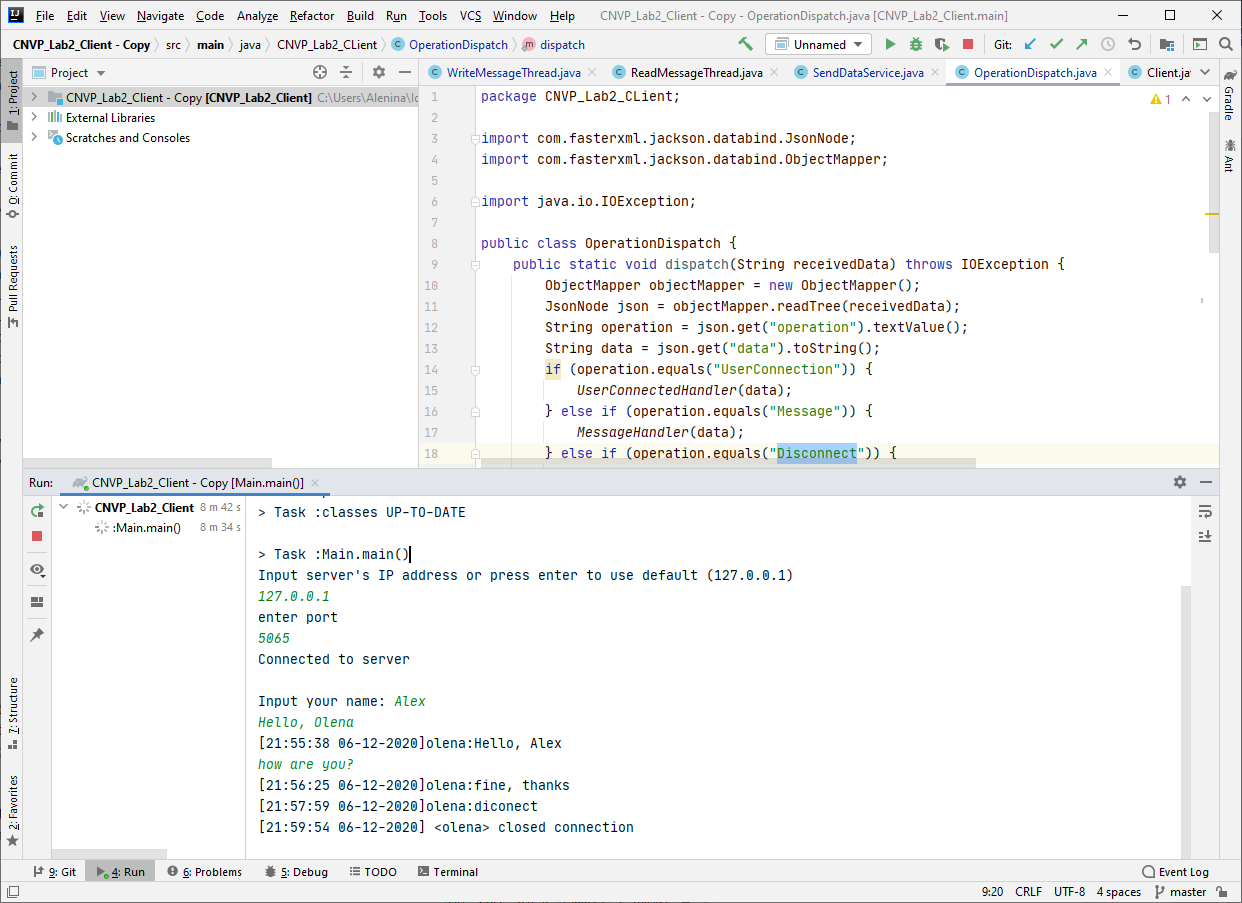
package CNVP\_Lab2\_CLient;  
  
import java.io.\*;  
import java.net.SocketException;  
  
public class ReadMessageThread extends Thread {  
  
 @Override  
 public void run() {  
 try {  
 while (true) {  
 if (Client.*isShutdownRequested*) {  
 break;  
 }  
  
 String received = Client.*reader*.readLine();  
 if (received == null) {  
 Client.*isShutdownRequested* = true;  
 break;  
 }  
 OperationDispatch.*dispatch*(received);  
 }  
 } catch (SocketException exception) {  
 System.*out*.println("Server closed connection");  
 Client.*isShutdownRequested* = true;  
 } catch (IOException exception) {  
 exception.printStackTrace();  
 }  
 }  
}

е) У методі main викликаємо метод Client.*run*.

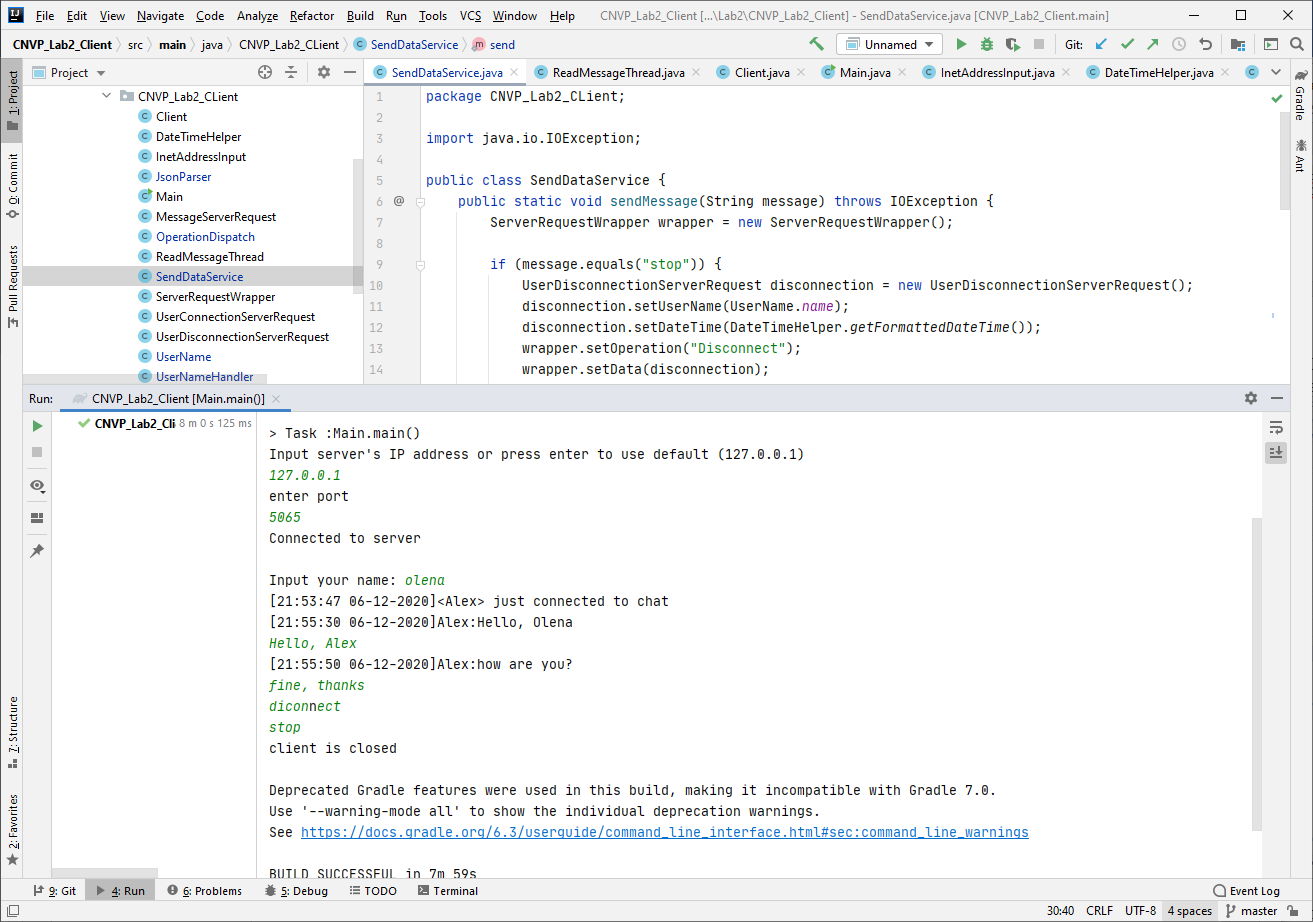
**Лістинг Main:**

package CNVP\_Lab2\_CLient;  
  
public class Main {  
  
 public static void main(String[] args) {  
 Client.*run*();  
 }  
}

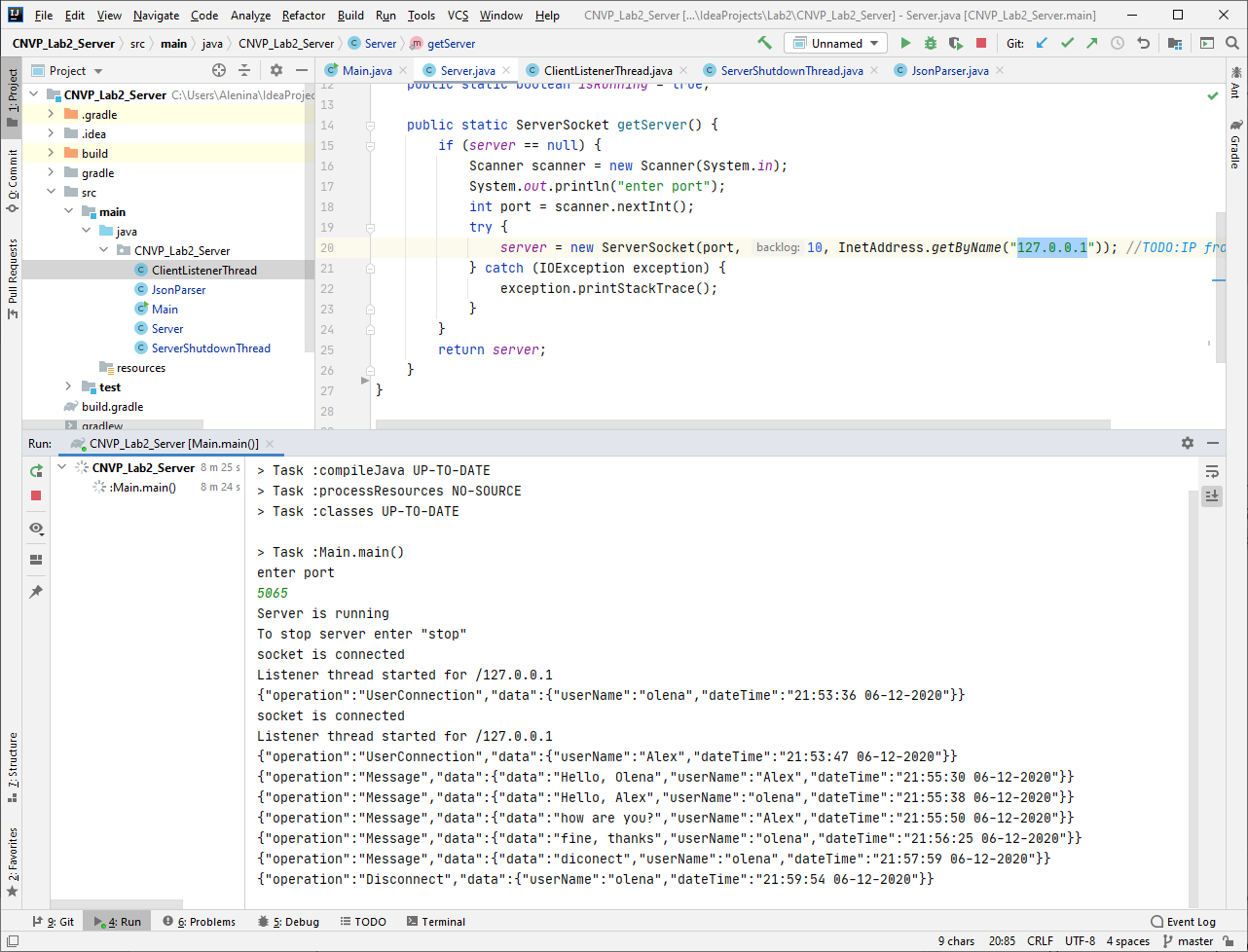
3. Результат роботи клієнта. Створюємо два клієнти та запускаємо одночасно. Результат роботи першого клієнта:



Результат роботи другого клієнта:

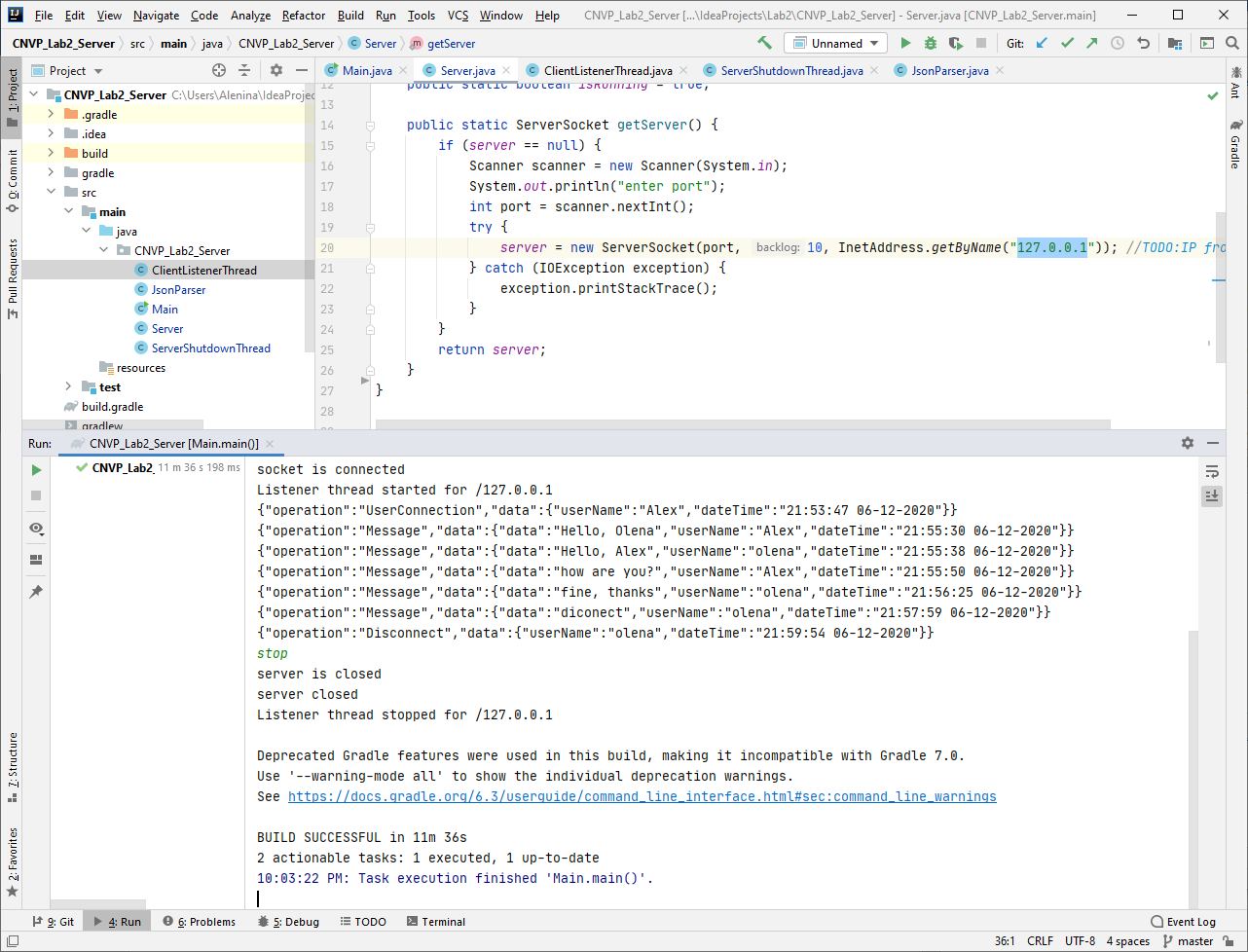


4. Результат роботи сервера:

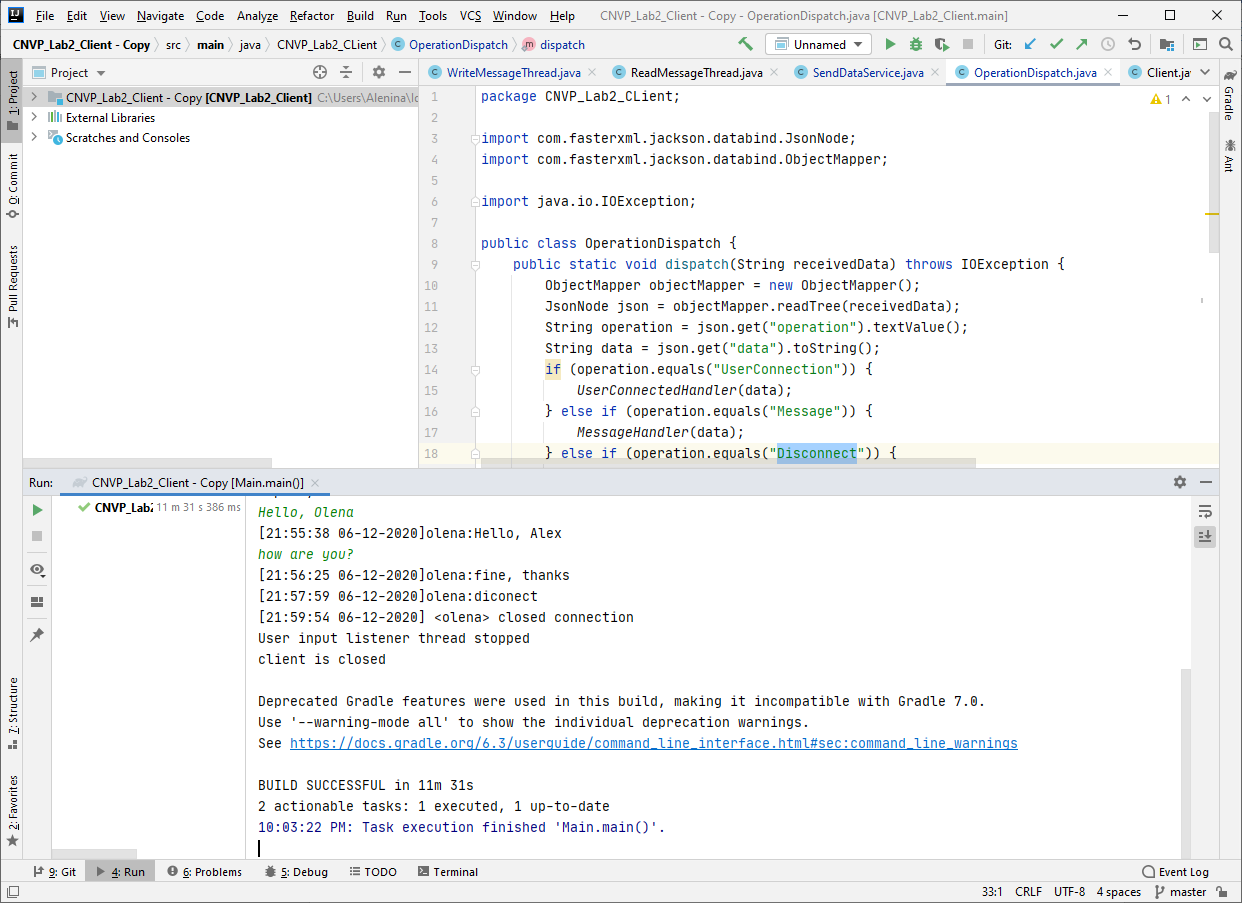


5. Відключення сервера.

а) Результат роботи на сервері, якщо сервер розірвав з’єднання:



б) Результат роботи на клієнті, якщо сервер розірвав з’єднання:



**Висновки:**

Під час виконання лабораторної роботи я ознайомилась з принципами роботи TCP сокетів, а також потоків (thread) та створила багатопотоковий клієнт-серверний застосунок.