Freie Universität Berlin

Prof. Dr. Margarita Esponda

# Nichtsequentielle Programmierung, SoeSe 2017

## Übungsblatt 8

**TutorIn: Lilli Walter**
**Tutorium 6**

Boyan Hristov, Sergelen Gongor

4. Juli 2017

---

Link zum Git Repository: https://github.com/BoyanH/FU-Berlin-ALP4/tree/master/Solutions/Homework8

## 1. Aufgabe

a) TCP Echo Server

```
1  import java.net.*;
2  import java.io.*;
3
4  public class TCPEchoServer {
5
6      public static void main(String[] args) {
7
8          // not as it should be done, I hope enough for this exercise. Usually each
       error should be handled individually
9          try {
10
11             int port = 1337;
12             String ipAddress = "localhost";
13             SocketAddress address = new InetSocketAddress(ipAddress, port);
14             ServerSocket socket = new ServerSocket(); // create unbound server socket
15             socket.bind(address); // bind it to the given address (ip + port)
16
17             System.out.println("TCP server running on localhost:1337");
18
19             // keep accepting client connections
20             while (true) {
21                 Socket connection = socket.accept(); // wait for client connections
       and accept them
22                 DataOutputStream out = new DataOutputStream(connection.getOutputStream
       ());
23                 BufferedReader br = new BufferedReader(new InputStreamReader(
       connection.getInputStream()));
24                 String currentLine = br.readLine();
25
26                 System.out.print("Received message \"");
27                 // read each line from the client and write it back
```

```
                while (currentLine != null) {
                    System.out.print(currentLine);
                    out.writeBytes(currentLine + '\n');
                    currentLine = br.readLine();
                }
                System.out.println("\" and wrote it back to client!");

                // close all streams and connections when finished
                out.close();
                br.close();
                connection.close();
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

b) TCP Echo Client

```
import java.net.*;
import java.io.*;

public class TCPEchoClient {

    public static void main(String[] args) {

        try {

            int port = 1337;
            String ipAddress = "localhost";
            Socket socket = new Socket(ipAddress, port); // create a client socket and bind it to our ServerSocket

            // Define all required IO streams and variables required for reading/writing from/to server
            DataOutputStream out;
            BufferedReader stdin;
            BufferedReader serverIn;
            String currentLine;
            String serverResponseLine;

            System.out.println("TCP client connected to localhost:1337"); // after new Socket() call has executed, client should be connected

            // initialize IO buffers
            out = new DataOutputStream(socket.getOutputStream());
            stdin = new BufferedReader(new InputStreamReader(System.in));
            serverIn = new BufferedReader(new InputStreamReader(socket.getInputStream()));

            System.out.print("Message to send to server: ");
            currentLine = stdin.readLine(); // read from console (stdin)
            out.writeBytes(currentLine + '\n'); // write to server
            serverResponseLine = serverIn.readLine(); // read response from server
            System.out.printf("Server response: %s\n", serverResponseLine);

            // close all buffers and socket connection
            out.close();
            stdin.close();
            serverIn.close();
            socket.close();



        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
    }
}
```

c) UDP Echo Server

```
import java.net.*;
import java.io.*;

public class UDPEchoServer {

    public static void main(String[] args) {

        // not as it should be done, I hope enough for this exercise. Usually each error should be handled individually
        try {

            int port = 1337;
            InetAddress laddr = InetAddress.getLocalHost();
            byte[] inDataBuffer = new byte[256];
            DatagramSocket udSocket = new DatagramSocket(1337, laddr);

            DatagramPacket udClientPacket;
            DatagramPacket udResponsePacket;
            int clientPort;
            InetAddress clientAddress;
            String clientMessage;

            System.out.println("Started UDP server on localhost:1337");

            // keep accepting client connections
            while (true) {
                udClientPacket = new DatagramPacket(inDataBuffer, inDataBuffer.length);
                udSocket.receive(udClientPacket); // wait to receive client user-datagramm-packet
                clientMessage = new String(udClientPacket.getData()); // get String from send byte[]
                clientAddress = udClientPacket.getAddress();
                clientPort = udClientPacket.getPort();

                udResponsePacket = new DatagramPacket(inDataBuffer, inDataBuffer.length, clientAddress, clientPort);
                udSocket.send(udResponsePacket);

                System.out.printf("Received message \"%s\" and wrote it back to client.", clientMessage);

                // nothing to close with UDP (except for the server socket, in our case NEVER! ^^)
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

d) UDP Echo Client

```
import java.net.*;
import java.io.*;

public class UDPEchoClient {

    public static void main(String[] args) {
```

```
        // not as it should be done, I hope enough for this exercise. Usually each error should be handled individually
        try {

            int port = 1337;
            InetAddress laddr = InetAddress.getLocalHost();
            byte[] responseDataBuffer = new byte[256];
            byte[] sendBuffer;
            DatagramSocket udSocket = new DatagramSocket(); // create new unbound datagramm socket

            DatagramPacket udClientPacket;
            DatagramPacket udResponsePacket = new DatagramPacket(responseDataBuffer, responseDataBuffer.length);
            BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));

            // read from stdin
            System.out.print("Enter message to send to server: ");
            sendBuffer = stdin.readLine().getBytes();

            // create a DatagramPacket from the read message and send to server
            udClientPacket = new DatagramPacket(sendBuffer, sendBuffer.length, laddr, port);
            udSocket.send(udClientPacket);

            // read server response
            udSocket.receive(udResponsePacket);
            System.out.printf("Server response: %s\n", new String(udResponsePacket.getData()));

            udSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

# Aufgabe 2

Wir haben alle Aufgaben für das TCP Chat bearbeitet, alle Features sollen funktionieren.

1. Client

```
package network;

import fx.ClientGUI;
import javafx.scene.paint.Color;

import java.io.IOException;
import java.io.PrintWriter;
import java.net.Socket;

public class ClientTCP extends AbstractChatClient {

    private MessageListenerTCP messageListenerTCP;
    private boolean clientConnected;
    private Socket serverSocket;
    private PrintWriter socketWriter;

    public ClientTCP(ClientGUI gui) {
        super(gui);
    }
```

```
21      @Override
22      public void sendChatMessage(String msg) {
23          if (this.socketWriter != null) {
24              this.socketWriter.println(msg);
25          }
26      }
28      @Override
29      public void connect(String address, String port) {
30          if (this.clientConnected) {
31              return;
32          }
34          boolean connected = true;
35          int portNumber;
37          try {
39              portNumber = Integer.parseInt(port);
40              this.serverSocket = new Socket(address, portNumber);
41              this.socketWriter = new PrintWriter(this.serverSocket.getOutputStream(),
    true); // auto-flush output stream
42              this.connectToChat();
44              this.messageListenerTCP = new MessageListenerTCP(this.serverSocket, this);
45              this.messageListenerTCP.start();
48          } catch (NumberFormatException e) {
49              this.gui.pushChatMessage("System: Port must be a valid integer!");
50              connected = false;
51          } catch (IOException e) {
52              this.terminate();
53          }
55          this.clientConnected = connected;
56      }
58      @Override
59      public void disconnect() {
60          this.terminate();
61      }
63      @Override
64      public void terminate() {
65          this.setConnected(false);
67          if (this.messageListenerTCP != null) {
68              this.messageListenerTCP.terminate();
69          }
70          try {
71              if (this.serverSocket != null) {
72                  this.serverSocket.close();
73              }
74              if (this.socketWriter != null) {
75                  this.socketWriter.close();
76              }
77          } catch (IOException e) {
78              e.printStackTrace();
79          }
80      }
82      @Override
83      public void setUName(String name) {
84          super.setUName(name);
86          // notify server
```

```
87          this.socketWriter.printf("/n %s\n", this.getUName());
88      }
90      public void setConnected(boolean connected) {
91          this.clientConnected = connected;
92          this.gui.setSymbolColor(this.clientConnected ? Color.GREEN : Color.RED);
93      }
95      public void onServerMessage(String message) {
96          if (message.charAt(0) == '/') {
97              this.handleServerCommand(message);
98          } else {
99              this.gui.pushChatMessage(message);
100         }
101     }
103     public void connectToChat() {
104         if (this.socketWriter != null) {
105             this.socketWriter.printf("/n %s\n", this.getUName());
106             this.socketWriter.flush();
107             this.setConnected(true);
108         }
109     }
111     private void handleServerCommand(String command) {
112         switch(command.substring(0,2)) {
113             case "/r":
114                 int firstSpaceIdx = command.indexOf(' ');
115                 int secondSpaceIdx = command.indexOf(' ', firstSpaceIdx + 1);
116                 this.gui.pushChatMessage(
117                         String.format("%s renamed to %s",
118                                 command.substring(firstSpaceIdx + 1, secondSpaceIdx),
119                                 command.substring(secondSpaceIdx + 1)
120                         )
121                 );
122                 break;
123             default:
124                 System.out.println("Unrecognized server command!");
125         }
126     }
127 }
```

```
1  package network;
3  import fx.ClientGUI;
4  import javafx.scene.paint.Color;
6  import java.io.BufferedReader;
7  import java.io.IOException;
8  import java.io.InputStreamReader;
9  import java.net.*;
11 public class MessageListenerTCP extends Thread {
13     private final int MAX_CHAT_REFRESH_LENGTH = 4096;
14     private String serverAddress;
15     private int serverPort;
16     private Socket serverSocket;
17     private boolean running = true;
18     private ClientTCP client;
19     private BufferedReader socketBuffer;
22     public MessageListenerTCP(Socket serverSocket, ClientTCP client) throws
       NumberFormatException {
23         this.serverSocket = serverSocket;
24         this.client = client;
```

```
26          try {
27              this.socketBuffer = new BufferedReader(new InputStreamReader(this.
    serverSocket.getInputStream()));
28              this.socketBuffer.mark(MAX_CHAT_REFRESH_LENGTH);
29          } catch (IOException e) {
30              this.terminate();
31              return;
32          }
33      }
34
35      public BufferedReader getReader() {
36          return this.socketBuffer;
37      }
38
39      public void terminate() {
40          this.running = false;
41          this.client.setConnected(false);
42      }
43
44      public Socket getServerSocket() {
45          return this.serverSocket;
46      }
47
48      @Override
49      public void run() {
50          this.waitForMessages();
51      }
52
53      /**
54       * Waits for server to send a message. If the new message is null, then the server
       has close the
55       * connection and we need to notify the client. If a new message is received, call
       ClientTCP::onServerMessage()
56       */
57      public void waitForMessages() {
58          String inputLine;
59
60          try {
61              this.serverSocket.setSoTimeout(1000);
62          } catch (SocketException e) {
63              e.printStackTrace();
64          }
65
66          while (this.running) {
67              try {
68                  inputLine = socketBuffer.readLine();
69                  if (inputLine == null) {
70                      // server not available
71                      this.terminate();
72                  } else {
73                      this.client.onServerMessage(inputLine);
74                  }
75              } catch (SocketTimeoutException e) {
76                  // timed out, server is there, has nothing to say
77              } catch (IOException e) {
78                  this.terminate();
79              }
80          }
81      }
82  }
```

2. Server

```
1  package network;
2
3  import fx.ServerGUI;
```

```java
import javafx.scene.paint.Color;

import java.io.BufferedReader;
import java.io.IOException;
import java.net.*;
import java.util.LinkedList;
import java.util.List;

public class ServerTCP extends AbstractChatServer {

    private boolean running = false;
    private ServerSocket serverSocket;
    private ConnectionAccepterTCP connectionAccepter;
    private List<ClientCommunicationThread> clientMessagers;

    public ServerTCP(ServerGUI gui) {
        super(gui);
        this.clientMessagers = new LinkedList<ClientCommunicationThread>();
    }

    @Override
    public void receiveConsoleCommand(String command, String msg) {
        switch(command) {
            case "/exclude":
                this.removeClientByName(msg);
                break;
            case "/mute":
                String[] words = msg.split(" ");
                if (words.length < 2) {
                    this.gui.pushConsoleMessage("Invalid number of arguments for command /mute");
                    return;
                }
                this.muteClientByNameForSeconds(words[0], words[1]);
                break;
            default:
                this.gui.pushConsoleMessage(String.format("Unrecognized console command %s", command));
        }

    }

    @Override
    public void start(String port) {
        SocketAddress address;
        int portNumber;

        if (this.running) {
            return;
        }

        try {
            portNumber = Integer.parseInt(port);
        } catch (NumberFormatException e) {
            System.out.println("Port must be an integer!");
            return;
        }
        address = new InetSocketAddress("localhost", portNumber);
        try {
            this.serverSocket = new ServerSocket(); // create unbound server socket
            this.serverSocket.bind(address);
            this.running = true;
            this.gui.setSymbolColor(Color.GREEN);
            this.connectionAccepter = new ConnectionAccepterTCP(this.serverSocket, this);
            this.connectionAccepter.start();

        } catch (IOException e) {
```

```
69              e.printStackTrace();
70              this.terminate();
71              return;
72          }

74      }

76      @Override
77      public void stop() {
78          this.running = false;
79          this.gui.setSymbolColor(Color.RED);

81          if (this.connectionAccepter != null) {
82              this.connectionAccepter.terminate();
83          }

85          try {
86              this.serverSocket.close();
87          } catch (IOException e) {
88              e.printStackTrace();
89          }

91          this.removeAllClients();

93      }

95      @Override
96      public void terminate() {
97          this.stop();

99      }

101     public void addNewClient(Socket connection) {
102         ClientCommunicationThread newClientThread = new ClientCommunicationThread(
        connection, this);
103         newClientThread.start();
104         this.clientMessagers.add(newClientThread);
105     }

107     public void removeClient(ClientCommunicationThread client) {
108         this.clientMessagers.remove(client);
109         this.gui.removeClient(client.getClientId());
110         client.terminate();
111     }

113     public void removeAllClients() {
114         while(!this.clientMessagers.isEmpty()) {
115             ClientCommunicationThread crnt = this.clientMessagers.remove(0);
116             this.gui.removeClient(crnt.getClientId());
117             crnt.terminate();
118         }
119     }

121     public void onNewMessage(ClientCommunicationThread clientMessager, String message)
         {
122         if (clientMessager.mutedBefore > System.currentTimeMillis()) {
123             clientMessager.getWriter().println("You are muted!");
124             clientMessager.getWriter().flush();
125             return; // muted
126         }

128         for (ClientCommunicationThread client : this.clientMessagers) {
129             client.getWriter().printf("%s: %s\n", clientMessager.getUName(), message);
130             client.getWriter().flush();
131         }
132     }

134     public void addClientToGUI(int id, String uname) {
```

```
135         this.gui.addClient(id, uname);
136     }
138     public void clientRenamed(int id, String oldName, String newName) {
139         this.gui.removeClient(id);
140         this.gui.addClient(id, newName);
142         // better send not as direct message but as command, wanted to
143         // refresh all mesages, but the gui windows is not flushable ;/
144         for (ClientCommunicationThread client : this.clientMessagers) {
145             client.getWriter().printf("/r %s %s\n", oldName, newName);
146             client.getWriter().flush();
147         }
148     }
150     public void sendPrivateMessage(ClientCommunicationThread fromClient, String
    toClientName, String message) {
151         fromClient.getWriter().printf("*private* %s: %s\n", fromClient.getUName(),
    message);
152         fromClient.getWriter().flush();
153         this.getClientByName(toClientName).getWriter().printf("*private* %s: %s\n",
    fromClient.getUName(), message);
154         this.getClientByName(toClientName).getWriter().flush();
155     }
157     private ClientCommunicationThread getClientByName(String name) {
158         for (ClientCommunicationThread client : this.clientMessagers) {
159             if (client.getUName().equals(name)) {
160                 return client;
161             }
162         }
164         return  null;
165     }
167     private void removeClientByName(String name) {
168         this.removeClient(this.getClientByName(name));
169     }
171     private void muteClientByNameForSeconds(String name, String seconds) {
172         int milliseconds;
174         try {
175             milliseconds = Integer.parseInt(seconds) * 1000;
176             this.getClientByName(name).mutedBefore = System.currentTimeMillis() +
    milliseconds;
177         } catch (NumberFormatException e) {
178             this.gui.pushConsoleMessage("Invalid argument for seconds to mute parsed!"
    );
179         }
180     }
182 }
```

```
1  package network;
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.io.InputStreamReader;
6  import java.net.ServerSocket;
7  import java.net.Socket;
8  import java.net.SocketException;
9  import java.net.SocketTimeoutException;
10 import java.util.LinkedList;
12 public class ConnectionAccepterTCP extends Thread {
14     private ServerSocket serverSocket;
```

```
    private ServerTCP server;
    private boolean running = true;
    private LinkedList<Socket> connections;

    public ConnectionAccepterTCP(ServerSocket socket, ServerTCP server) {
        this.serverSocket = socket;
        this.server = server;
        this.connections = new LinkedList<>();
    }

    @Override
    public void run() {
        this.waitForConections();
    }

    public void waitForConections() {
        String inputLine;
        BufferedReader socketBuffer = null;

        while (this.running) {
            try {
                Socket connection = this.serverSocket.accept(); // wait for client
    connections and accept them

                if (!this.connections.contains(connection)) {
                    this.server.addNewClient(connection);
                    this.connections.add(connection);
                }

            } catch (IOException e) {
                // socket setTimeout exception, whatever
            }
        }
    }

    public void terminate() {
        this.running = false;
    }
}
```

```
package network;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.SocketException;
import java.net.SocketTimeoutException;

public class ClientCommunicationThread extends Thread {

    static int clientsCount = 0;

    private int id;
    private Socket clientConnection;
    private String clientName;
    private boolean running = true;
    private ServerTCP server;
    private PrintWriter socketWriter;
    public long mutedBefore = System.currentTimeMillis();

    public ClientCommunicationThread(Socket connection, ServerTCP server) {
        this.id = ++clientsCount;
        this.clientConnection = connection;
        this.server = server;
        try {
            this.socketWriter = new PrintWriter(this.clientConnection.getOutputStream
```

```
    (), true); // auto-flush output stream
29          } catch (IOException e) {
30              e.printStackTrace();
31              this.terminate();
32          }
33      }

35      public String getUName() {
36          return this.clientName;
37      }

39      public void setUName(String name) {
40          if (this.clientName == null) {
41              this.server.addClientToGUI(this.id, name);
42          } else {
43              this.server.clientRenamed(this.getClientId(), this.clientName, name);
44          }
45          this.clientName = name;
46      }

48      public PrintWriter getWriter() {
49          return this.socketWriter;
50      }

52      public int getClientId() {
53          return this.id;
54      }

56      @Override
57      public void run() {
58          this.waitForMessages();
59      }

61      public void terminate() {
62          this.running = false;
63          try {
64              this.clientConnection.close();
65              this.socketWriter.close();
66          } catch (IOException e) {
67              e.printStackTrace();
68          }

70      }

72      public void waitForMessages() {
73          String inputLine;
74          BufferedReader socketBuffer;

76          try {
77              socketBuffer = new BufferedReader(new InputStreamReader(this.
    clientConnection.getInputStream()));
78              socketBuffer.mark(4096);
79          } catch (IOException e) {
80              e.printStackTrace();
81              this.terminate();
82              return;
83          }

85          try {
86              this.clientConnection.setSoTimeout(1000);
87          } catch (SocketException e) {
88              e.printStackTrace();
89          }

91          while (this.running) {
92              try {
93                  inputLine = socketBuffer.readLine();
94                  if (inputLine == null) {
```

```
                    // client disconnected
                    this.server.removeClient(this);
                } else if (inputLine.charAt(0) == '/') {
                    this.parseClientCommand(inputLine);
                } else {
                    this.server.onNewMessage(this, inputLine);
                }
            } catch (SocketTimeoutException e) {
                // timed out, client is there, has nothing to say
            } catch (IOException e) {
                this.server.removeClient(this);
            }
        }

        try {
            socketBuffer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void parseClientCommand(String clientMessage) {;
        int firstSpaceIndex = clientMessage.indexOf(' ');
        String command = clientMessage.substring(0, firstSpaceIndex);
        String rest = clientMessage.substring(firstSpaceIndex+1);

        switch(command) {
            case "/n":
                this.setUName(rest);
                break;
            case "/private":
                this.handlePrivateMessage(rest);
            default:
                // unrecognized command
                break;
        }
    }

    private void handlePrivateMessage(String message) {
        int firstSpaceIdx = message.indexOf(' ');
        String uName = message.substring(0, firstSpaceIdx);
        String privateMessage = message.substring(firstSpaceIdx+1);
        this.server.sendPrivateMessage(this, uName, privateMessage);
    }
}
```