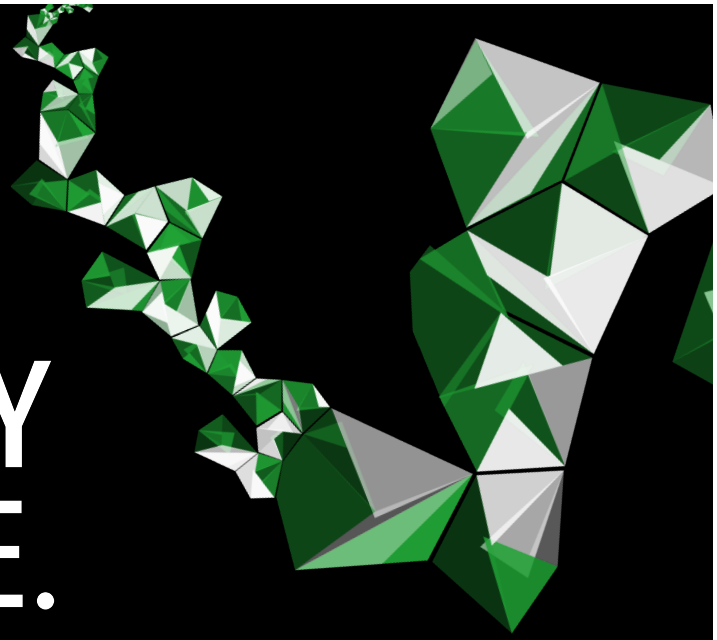
FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND COMPUTER SCIENCE (EEMCS)



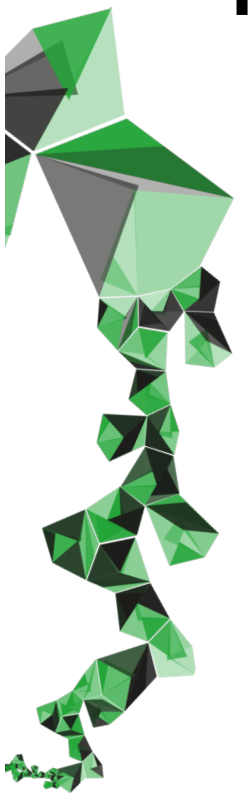
UNIVERSITY OF TWENTE.

P7.2: NETWORKING AND MULTITHREADING

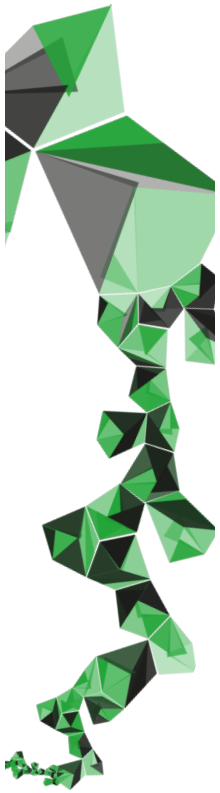
PROGRAMMING (202001066) | MODULE 2: SOFTWARE DEVELOPMENT (202001064)

5 JANUARY 2021

PROGRAMMING LINE OVERVIEW



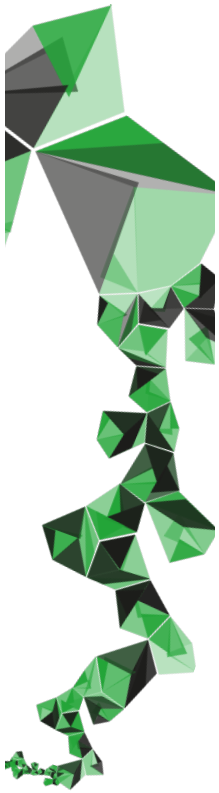
Week 1 Values and variables Control flow	**Week 2** Classes and objects Testing	**Week 3** Interfaces and Inheritance Subtyping Security 1
Week 4 Arrays and Lists List implementations Collections	**Week 5** Exceptions Stream I/O and MVC Security 2	**Week 6** Concurrency Project kick-off IDE Tips & Tricks
Week 7 Basic Networking **Networking and Multithreading** GUIs	**Week 8/9** Advanced Java facilities Test	**Week 10** Project Test resit



CLIENT/SERVER APPLICATIONS
MORE EXAMPLES

- Until now in our Webcat and DateClient/DateServer examples a client queries a server
- Alternative: `CLChatClient` and `CLChatServer` from Eck
  - After connection is established, client and server behave the same (are symmetric)
  - Server and client perform a handshake and send and receive messages
  - I modified the code to stress symmetry by defining a `CLChatHandler` class that is shared by Client and Server

CLCHATAPP



CLChatHandler

- Methods `doHandshake`, `sendMessage` and `receiveMessage`

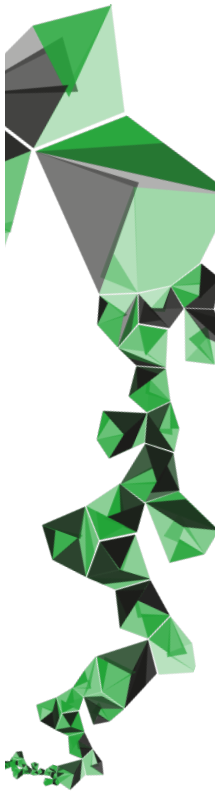
Server

- Listens to a client connection request on port 1728 (default)
- After connection is established creates a `CLChatHandler` and performs handshake (call `doHandshake`)
- In a loop, receives and sends messages with the `CLChatHandler`

CLCHATSERVER

```
70          /*
71           * Wait for a connection request. When it arrives, close down the listener.
72           */
73          try {
74              listener = new ServerSocket(port);
75              System.out.println("Listening on port " + listener.getLocalPort());
76              connection = listener.accept();
77              listener.close();
78          } catch (IOException e) {
79              System.out.println("An error occurred while opening connection.");
80              System.out.println(e.toString());
81          }
82          /*
83           * Create a handler to do the handshake and exchange messages.
84           */
85          try {
86              CLChatHandler handler = new CLChatHandler(connection);
87              handler.doHandshake();
88              System.out.println("Connected.  Waiting for the first message.");
89              while (true) {
90                  handler.receiveMessage();
91                  handler.sendMessage();
92              }
93          } catch (IOException e) {
94              System.out.println("Sorry, an error has occurred.  Connection lost.");
95              System.out.println("Error:  " + e);
96          }
97      }
```

CLCHATAPP



Client

- Starts connection on port 1728
- After connection is started, creates a `CLChatHandler` and performs handshake (call `doHandshake`)
- In a loop, sends and receives messages with the `CLChatHandler`

CLCHATCLIENT

```
88          /*
89           * Open a connection to the server.
90           */
91          try {
92              System.out.println("Connecting to " + computer + " on port " + port);
93              connection = new Socket(computer, port);
94          } catch (IOException e) {
95              System.out.println("An error occurred while opening connection.");
96              System.out.println(e.toString());
97          }
98          /*
99           * Create a handler to do the handshake and exchange messages.
100          */
101         try {
102             CLChatHandler handler = new CLChatHandler(connection);
103             handler.doHandshake();
104             System.out.println("Connected!!! Enter your first message.");
105
106             while (true) {
107                 handler.sendMessage();
108                 handler.receiveMessage();
109             }
110         } catch (Exception e) {
111             System.out.println("Sorry, an error has occurred.  Connection lost.");
112             System.out.println("Error:  " + e);
113         }
114     }
```

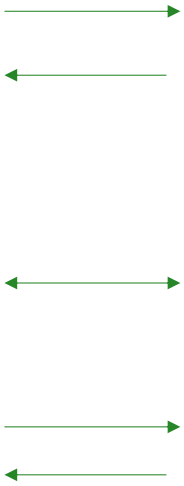
CLCHATAPP
END-TO-END COMMUNICATION

CLChatClient

```
request connection
(port 1728);
when connection is
confirmed, create
CLChatHandler;

DoHandshake;

while (true) {
 sendMessage;
 receiveMessage;
}
```



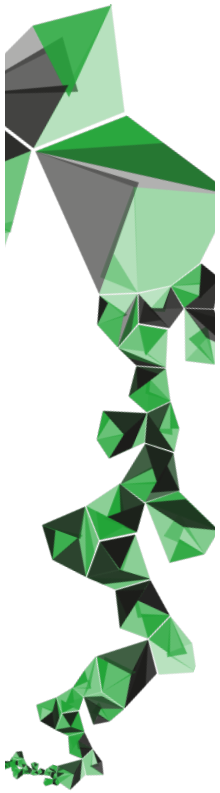
CLChatServer

```
listen on port 1728;
when connection is
accepted, create
CLChatHandler;


DoHandshake;

while (true) {
 receiveMessage;
 sendMessage;
}
```

PROBLEM WITH CLCHATAPP



- Client and Server must wait for each other to send and receive messages!
- Unrealistic since network communication is inherently asynchronous

  → No idea when messages are sent (and must be received)
- How to cope with asynchrony?

  → Multithreading: different threads for network and user inputs!

BLOCKING AND I/O

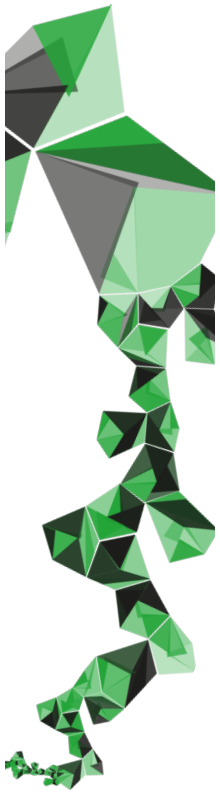
- Program that reads data from input stream, waits (blocks) until data are available, or returns immediately if data were sent before
- Example: Date client

```
35          try {
36              connection = new Socket(hostName, LISTENING_PORT);
37              incoming = new BufferedReader(new InputStreamReader(connection.getInputStream()));
38              String lineFromServer = incoming.readLine();
39              if (lineFromServer == null) { // Null indicates end-of-stream
40                  throw new IOException("No data was sent!");
41              }
```

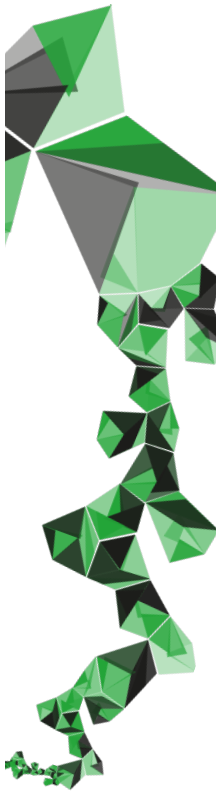
- Programs may also block when performing output! Why?

  Attempts to send faster than data can be transmitted!!!

BLOCKING AND MULTITHREADING

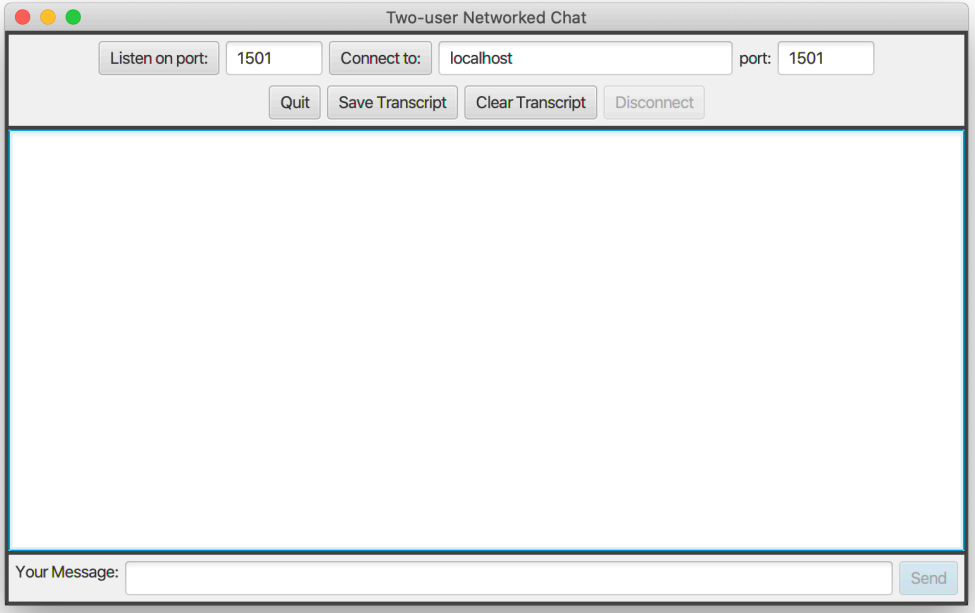


- If a program (thread) blocks it can only be unblocked when the event it is waiting for happens (e.g., data arrival)
- If the program needs to be reactive, for example, to the user interface, we need multiple concurrent threads so that some threads can be blocked while others keep on working
- Situation even worse with servers that must handle multiple clients!
- Best practice: after a connection is established, create a thread only to handle this connection



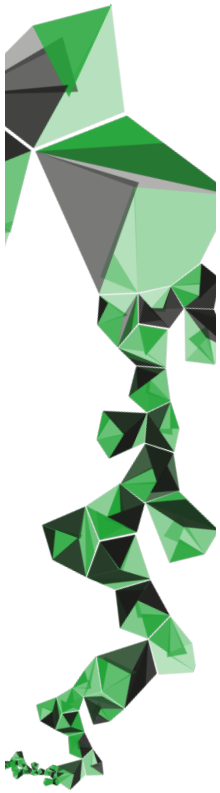
EXAMPLE: GUICHAT
NETWORKING CODE

- Works as client or server
- Start one program (‘server’), let it listen to a port and connect with another (‘client’)
- After connection is established, both `GUIChat` programs work in the same way



UNIVERSITY OF TWENTE.

CONNECTIONHANDLER CLASS



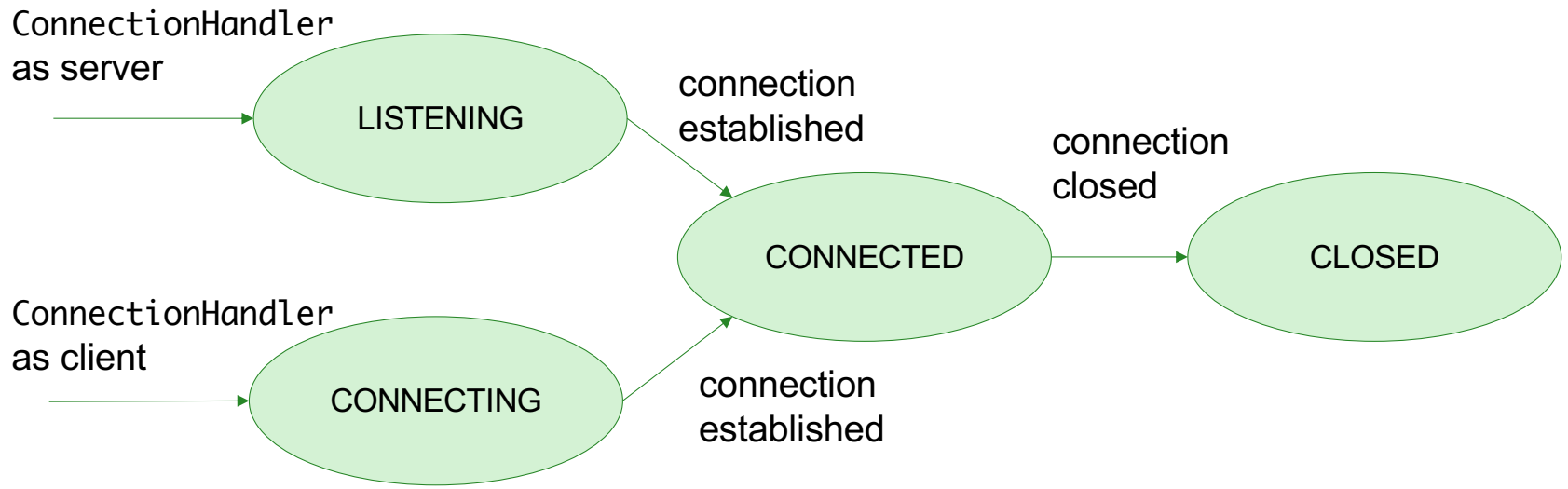
- A `ConnectionHandler` inner class extends `Thread` and has been defined to handle the chat protocol
- `ConnectionHandler` object is created and started when a `GUIChat` listens to a port (acts as server) or attempts to establish a connection (acts as client)

```
/**
 * Defines the thread that handles the connection.  T
 * for opening the connection and for receiving messa
 * several methods that are called by the main class,
 * executed in a different thread.  Note that by usir
 * connection, any blocking of the graphical user int
 * using a thread for reading messages sent from the
 * can be received and posted to the transcript asyn
 * time as the user is typing and sending messages.
 * that are made by this class are done using Platfor
 */
private class ConnectionHandler extends Thread {

    private volatile ConnectionState state;
    private String remoteHost;
    private int port;
    private ServerSocket listener;
    private Socket socket;
    private PrintWriter out;
    private BufferedReader in;

    /**
     * Listen for a connection on a specified port.
     * does not perform any network operations; it ju
     * instance variables and starts the thread.  Not
     * thread will only listen for one connection, ar
     * close its server socket.
     */
    ConnectionHandler(int port) {
        state = ConnectionState.LISTENING;
        this.port = port;
```

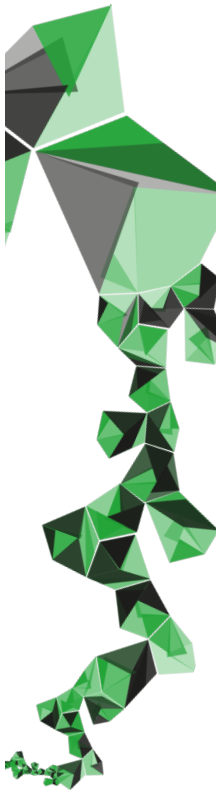
CONNECTION STATES



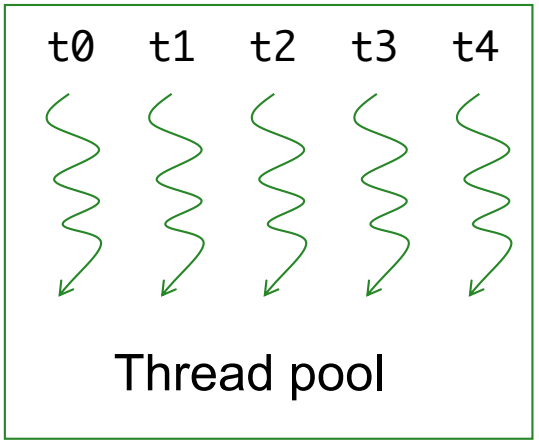

```
/**
 * Possible states of the thread that handles the network connection.
 */
private enum ConnectionState { LISTENING, CONNECTING, CONNECTED, CLOSED }
```


UNIVERSITY OF TWENTE.

THREAD POOL

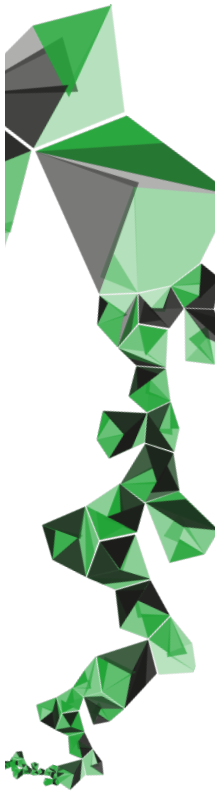


- Instead of creating threads on demand, we can create a pool of threads beforehand and allocate an available thread to a connection when necessary
- This can be used to avoid the thread creation overhead and limit the number of threads being created (avoid depleting resources)
- May be too sophisticated for this module...



See `DateServerWithThreadPool` example from Eck!

TAKE HOME MESSAGES



- Protocols define the rules for interoperability of system parts
- The client/server architecture is a quite popular for implementing distributed software systems
- Java offers support to build (client/server) applications on top of transport protocols like TCP
- For less than trivial applications, it is necessary to use multithreading, so that different connections can be supported while keeping the user interface responsive