Study Point Exercise-2 – Threads and TCP Programming

You can earn a maximum of **5** points for this exercise as outlined below:

- 1 point for the Exercise: Exam-preparation-1_ProducerConsumer.pdf
- Max 2 points for the Echo Server Exercise
- Max 2 points for exercise: "Upload the Simple Echo Server to Azure"

If you hand in via mail you can get the additional "attendance point" if your score is **three** our above. If it's not, you should probably have attended the class to get help ;-)

When to hand in

If you hand in via mail send a mail as described below no later than **Friday, September 4th. 15.00**

If you attend the class, you can demo your solution up until **15.45**

These deadlines are hard. The tutors are paid until 16.00 and are **not** expected to continue after that

How to hand in:

Either demonstrate what you did in the class or send a mail to iwantstudypoints@gmail.com including the following:

Topic: Study Point Exercise-2

Content:
First line should be your full name,
Second line: the link to your Git-hub repository for : Exam-preparation-1_ProducerConsumer.pdf.
Third line: the link to your Git-hub repository for the Echo Server Exercise
Fourth line: Instructions (ready to cut an paste) in how to connect to the server via Telnet

Exam Preparation Exercise – Producer Consumer

Solve the exercise "Exam-preparation-1_ProducerConsumer.pdf" in the folder examPreparation.

Simple Echo Server – Preparing for CA-1

This exercise will guide you through the steps in creating a very simple Echo server + client which will echo any string sent from a client back to (all) clients upper cased (hello world → HELLO WORLD). During the exercise you will be guided through a number of steps/hints, all required for CA-1.

The task is not to complete the exercise as fast as possible, but to understand the hints you are given throughout the exercise.

Part-1

- Clone the start code for the project: https://github.com/Lars-m/EchoStudyPointEx2.git
- Observe the following changes compared to what we did together:
  1. How protocol strings shared by both the server and client no longer are hard coded
  2. How the simple "test" in main is replaced by a **unit test** that automatically starts the server. Run the tests and verify that it is "green".
  3. For the server, all initialization values are placed in a **properties file** (in the root of the project)
  4. For the client initialization values are read from the command line
  5. Two scripts are added to the root folder; startServer.bat and startClient.bat
  6. The following element is added to the projects build.xml file:

```
<target name="-post-compile">
<copy file="server.properties" todir="${dist.dir}"/>
<copy file="startServer.bat" todir="${dist.dir}"/>
<copy file="startClient.bat" todir="${dist.dir}"/>
</target>
```

     It ensures that the two scripts and the server.properties file are copied into the **dist**-folder whenever you select "clean and build".
     The **dist** folder now includes everything necessary for deployment

Part-2 Provide the server with a log-file

When a new server goes into production, it is extremely important to have access to run-time log-information if anything goes wrong. Observe how Exceptions and general run-time information in the server is logged via statements like:

```
Logger.getLogger(EchoServer.class.getName()).log(Level.INFO, "Connected to a client");
```

Add the following statement to the start of the server's main method:

```
String logFile = properties.getProperty("logFile");
Utils.setLogFile(logFile,EchoServer.class.getName());
```

This will direct the log output to a file (logFile → given in the properties file).

Add a finally statement to the servers main method and add this code to ensure the file is closed correctly:

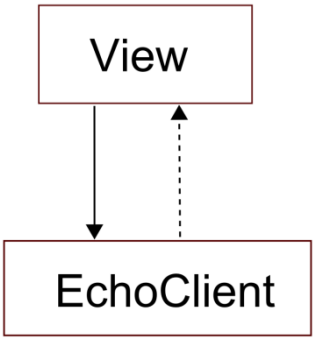
```
Utils.closeLogger(EchoServer.class.getName());
```

Test and **observe** the content in the generated log-file.

Part-3 Solving the Blocking Problems – Client Side

We are using a BLOCKING API (as opposed to for example Node.js) for our Network communication. This gives a number of challenges both for the client and the server.

Providing the client with a GUI



Rewrite the client so that it can be used by a GUI. Obviously this should be done the "right way" so we keep model and presentation separated.

This is pretty simple for the send(..) method since the Vew can create an EchoClient instance and just call the `send(..)` method.

For the receive method it's a bit more complex. The problem is that `receive()` contains a blocking call, so calling the method will block the GUI until we (if ever) receive something from the server. For this simple Echo example, we could probably live with having a call that called send() and receive() just after each other (as in the test) but what if we did not new when we were about to receive something (as in a chat program).

This problem is two-fold:

First, the view cannot call the EchoClient for information about received data. It is the EchoClient that should notify the View whenever it receives data. The `EchoClient` should not however, have any knowledge about any concrete View's.

This calls for the **Observer Pattern:** http://en.wikipedia.org/wiki/Observer_pattern

Secondly, we still have the problem with the blocking call, whenever we try to read from the network socket. This problem calls for **Threads**.

Implement an observer design in your client

Solve the first problem by implementing an observer based solution in the client.

You can do this in two ways, both introduced in exercises presented last week. Either implement everything by yourself, inspired by the observer pattern (https://en.wikipedia.org/wiki/Observer_pattern) or use the library interface + class:

- java.util.Observer (http://docs.oracle.com/javase/7/docs/api/java/util/Observer.html)
- java.util.Observable (http://docs.oracle.com/javase/7/docs/api/java/util/Observable.html)

Hints (for the second and recommended way):

- If you let the `EchoClient` class extend the `Observable` class introduced above, it will inherit all the methods presented in the Wikipedia's Subject class (see UML diagram in the Wikipedia article).
- You don't need to implement your own interface. Just let Observers (your GUI, or first, the test code in main) implement the Observer interface.
- Finally rename the `String receive()` method into void `run()` and add these two lines after the line that reads from the network stream:

```
setChanged();
notifyObservers(msg);
```

Change the" test" code in main to register as a listener, and call the run() method in the connect(..) call.

Did you get a response from the server?

The answer to the question above is probably **NO**. The `run()` method performs a blocking call and blocks the client completely until it receives data.

Change `EchoClient` to extend Thread and call `start()` instead of run(), this should fix the problem (why?).

Finally, before you actually design the GUI, change the unit test to reflect the new behaviour.

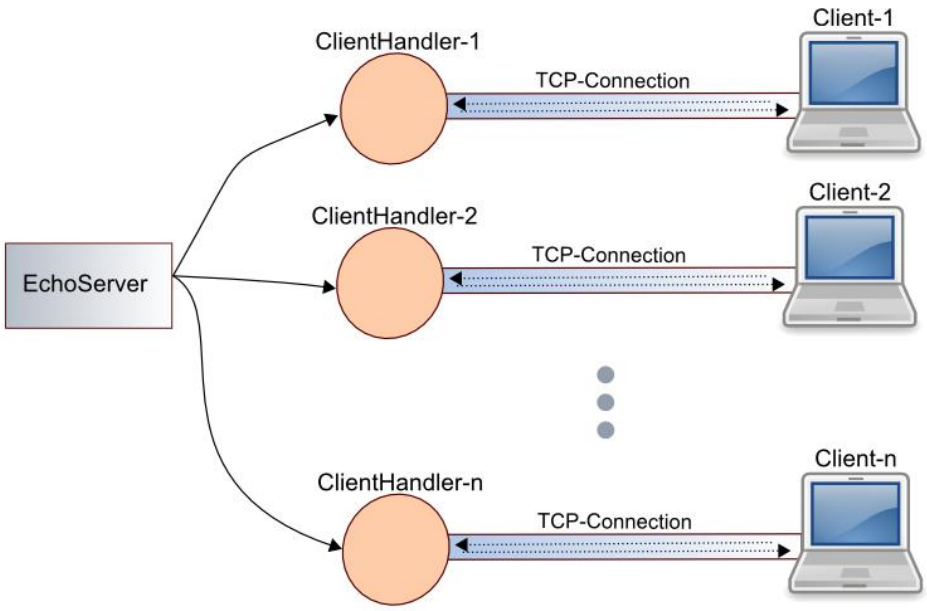
Add a simple GUI that uses the `EchoClient` to send and receive (via the `messageArrived(..)` method) to/from the server.

Part-4 Solving the Blocking Problems – Server Side

a) Change the **startClient.bat** script to start the GUI, instead of as originally, the EchoClient. Start the server and two or more instances of the client.

Notice how only the first client is being served (until it closes). The problem (again) is due to the blocking calls in the `handleClint(..)` method which never allow the outer loop to come back to the `accept()` method to serve a new client.

Again, this problem calls for Threads. What we would like is a design as sketched below where the EchoServer will spawn a new thread (look at it, as a virtual CPU) for each incoming connection.



b) Move the `handleClient`() method into a new class (remove the static declaration) called `ClientHandler` which should extend Thread (refer to the illustration on the previous page).

c) Change the method to become a constructor (that takes a Socket as the argument).

d) Move everything below the PrintWriter declaration into a method called `run()` (remember the class extends Thread)

e) Move the declaration of `input`, `writer` and the `socket` up to the top of the class to make them visible to both the constructor and the `run()` method.

f) Replace the original call (in EchoServer) to handle Client with a declaration of a new ClientHandler and call its start method.

g) Test that the server now has the ability to handle many clients "simultaneously".

Final additions, to prepare you for the design of the Chat system.

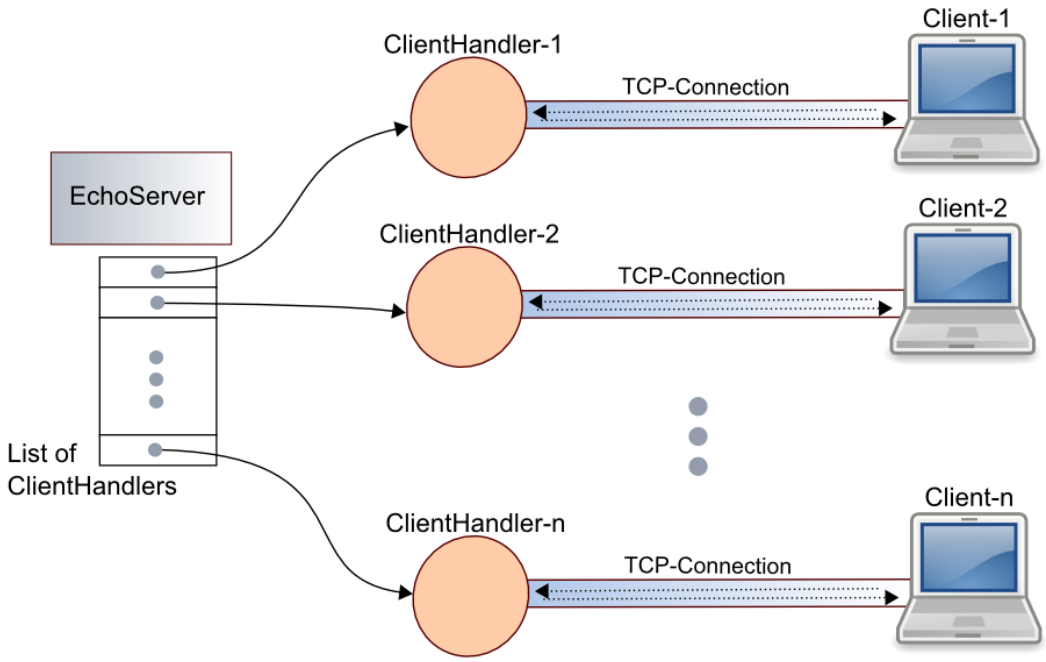
So far the server only echoes back a message to the client that sent the message. For a chat system a message must be delivered to either all, or specific clients other than the one that actually sent the message.

In this last part you should change the EchoServer and ClientHandler so that a message is delivered (upper cased) to **ALL** connected clients.

You could do this by following the steps given below:

- Add a list of `ClientHandlers` to the `EchoServer` and add all clients to the list when they connect.
- Provide the `EchoServer` with a `removeHandler(ClientHandler ch)` method, called by ClientHandlers when a connection is closed.
- Provide each `ClientHandler` with a `send(String message)` which should output the string to the handlers output stream (that is, send it to its actual client).
- Provide the `EchoServer` with a `send(String msg)` method which should iterate through all handlers and call their `send(..)` method with the argument upper cased.

This design is reflected in this figure



Upload the Simple Echo Server to Azure

Follow the steps in "To install a Java application server on your virtual machine" in the document javaOnAzureEcho.pdf, and install your server on Azure.

Verify that you access the server using the client implemented in part-3.