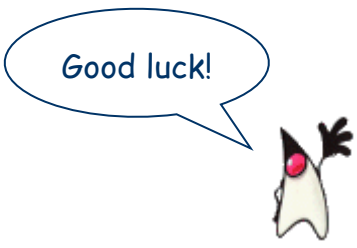
Mittuniversitetet
MID SWEDEN UNIVERSITY

Assignment 8

Java for C ++ programmers, 7,5 hp

Objective:	To write a multi-threaded client / server solution that communicates with each other through a Socket.
To read:	Lecture 9,10
Tasks:	3 (2 of them are optional)
Submission:	Inlämningslåda 8 at Moodle



Task 1

In this task, you will start with your previous solution from Lab 7. Create a new Java project in Eclipse, which will be a copy from your previous solution. Your classes should belong to the package: `dt062g.studentid.assignment8` where studentid is your username in the student portal/Moodle.

The classes which represents or uses a graphical user interface should be based on Swing. `System.in` may not be used for user input in these classes. `System.out` and `System.err` can only be used for debugging error messages. You may not expect the user of the program to see these messages.

Each class and interface (interface) that you create must be documented with comments. Ex.

```
/**
* A short description (in Swedish or English) of the class.
*
* @author  Your Name (your student id)
* @version 1.0
* @since   yyyy-mm-dd (last edited)
*/
```

You will create a client/server solution for your drawing program so that it can be possible to get/load and store/save drawings (XML files) from/to the server from the drawing program (client).

You will start by writing the server part and its classes, then you will write the client and finally adjust your `JFrame` so that it can use your client. All classes that consists of the server part should belong to the package:
`dt062g.studentid.assignment8.server`.

All classes that consists of client part should belong to the package:
`dt062g.studentid.assignment8.client`

Here studentid is your username in the student portal/Moodle

Server

Start by creating a new class called `Server`. You may decide by yourself whether the server should have a graphical user interface or if it should use the console. For console-based interface, it is ok to make messages to the user with `System.out` and `System.err`. This class should:

- Have a `main` method. In the `main` method, check if the number of arguments (`args.length`) given to the method are greater than 0. If so, try to use the first argument for the port which server should listen to. If no argument is given or the argument cannot be converted to an `int`, the server should listen to the port 10000 (the port may change in another assignment).

- Use a `ServerSocket` to wait for clients to connect to the server.
- Start a new thread for each connected client. In this thread (see description below), all communication between the client and the server should take place.
- The server should be able to handle multiple clients simultaneously.
- Show a message when the server is started and which port it is listening to.
- For each connected client, a message should be displayed together with which address it has joined from.
- When communication with a client is terminated for any reason, a message must be displayed that the connection with the client is closed.

```
Server started on port 10000
New client connected from 127.0.0.1:50081
New client connected from 127.0.0.1:50125
Client from 127.0.0.1:50081 has disconnected
New client connected from 127.0.0.1:50163
Client from 127.0.0.1:50163 has disconnected
Client from 127.0.0.1:50125 has disconnected
```

ClientHandler

This class will inherit from the `Thread` class. The class should:

- Have an instance variable of type `Socket`. Through this socket communication to/from the client will take place.
- Have a constructor that takes a `Socket` object as an argument. This socket will be assigned to the one above.
- Override the `run` method from the class `Thread`. In this method, all communication to/from the client should occur. Files are also being transferred here. It is ok that this method calls other methods in the class.
- Include other appropriate instance variables and methods that you think are necessary.

```
Server started on port 10000
New client connected from 127.0.0.1:52124
Command 'list' received from 127.0.0.1:52124
Sending list of files to 127.0.0.1:52124
Client from 127.0.0.1:52124 has disconnected
New client connected from 127.0.0.1:52125
Command 'load' received from 127.0.0.1:52125
Size of 'Circles by Robert.xml' is 1600 bytes
Sending file to 127.0.0.1:52125
File sent to 127.0.0.1:52125
Client from 127.0.0.1:52125 has disconnected
New client connected from 127.0.0.1:52126
Command 'save' received from 127.0.0.1:52126
Receiving 'Circles by Robert (copy).xml' from 127.0.0.1:52126
File received from 127.0.0.1:52126
Client from 127.0.0.1:52126 has disconnected
New client connected from 127.0.0.1:52127
Command 'list' received from 127.0.0.1:52127
Sending list of files to 127.0.0.1:52127
Client from 127.0.0.1:52127 has disconnected
```

You can decide by yourself how server and the client will communicate with each other. A possible solution can be that the client sends commands (`String`) on what should happen (list, load and save) and in the `run`-method you can adjusts the communication based on these commands. A command can consist of multiple operations. For example, the save command may consist of the following operations:

- Client sends the command save
- The server receives the command
- The client sends the name of the file to be saved
- The server receives the file name
- The client sends the file size (number of bytes)
- The server receives the file size
- The client sends the file content (convert the file into an array of `byte`)
- The server receives the file content and saves to a file on the server's hard disk according to the previously received file name (it is ok to overwrite existing files with the same name).

You will use `DataInputStream` and `DataOutputStream` to send commands and files and other information on both server and client side. It is important that you close these streams by calling `close` as soon as you no longer need them. Choose by yourself which streams you want to use to read/write files from/to hard drive.
It is important that you, if possible, nest the above streams in a buffered stream.

When the client and server are finished with a command, you should terminate the connection by calling `close` on all open streams and then `close` the `Socket` also. When it's time to perform a new command, the client will reconnect to the server again.

The server should store its XML files in a directory called xml. In this catalog, there may be files with an extension other than .xml. You must filter with a `FilenameFilter` so that only the XML files in this directory are sent to the client.

Client

You will write a class that will manage communication with the server. This class will be named `Client`. The class should be used completely separately (without the interference of other classes your drawing program have) to communicate with the server as described in the `Server` and `ClientHandler` classes. In addition the `ClientTest` class is also given. You will write your client so that it can be used by this test class without any changes.

The class must have at least the following:

- An instance variable of type `String` to store the address to the server.
- An instance variable of type `int` to store which port the server listens to.
- A constructor that takes an address (`String`) and port (`int`) as argument. It will be used while communicating with the server.

- A constructor without address that uses the default values "localhost" for address and 10000 for port.
- A method `connect` that connects to the server (creates a `Socket` to the server and the streams needed for communication). If the method is called and a connection is already established, no new connection will be made. The method should return `true` if socket/streams could be created and `false` if these could not be created.
- A method `disconnect` that terminates the connection with the server (closes and sets to `null` the socket and any streams that are used).
- A method `getFilenamesFromServer` that returns an array of strings (`String[]`) that contains the filenames of all XML files on the server in the xml folder. The method should begin by connecting to the server and end with closing the connection to the server (before returning). Let the method return `null` if any error occurred. Otherwise return an array of `String` of length 0 (zero) if the server does not have any files or an array of `String` that contains all the files on the server.
- A method `getFileFromServer` that takes a `String` as argument. This string contains the name of the file which will be retrieved from the server. The method should begin by connecting to the server and ending with interrupting the connection to the server (before returning). Retrieved means that the file should be copied from the xml folder on the server to the client's hard drive (in any folder, but not the same as the server). The method should return a string containing the entire path (on the client's hard drive) of the file copied (ex: "c:\tmp\Mona Lisa.xml"). If an error occurs, the method must return null. If the file does not exist on the server, an empty string of length 0 (zero) should be returned ("").
- A method `saveAsFileToServer` that takes two strings as argument. The first string contains the name of a local file to be sent to the server and saved there. The second string contains the name of the file to be saved on the server. The method should begin by connecting to the server and end with closing the connection to the server (before returning). Sent/saved means that the file should be copied from the client's hard disk to the xml folder on the server's hard drive. The method should return `true` if everything went well and `false` if something went wrong.
- A method `saveFileToServer` that takes one string as argument. This string contains the name of a local file to be sent to the server saved there. On the server, the file must be saved under the same name as the original file in the xml folder. This method should then work in the same way as `saveAsFileToServer`.

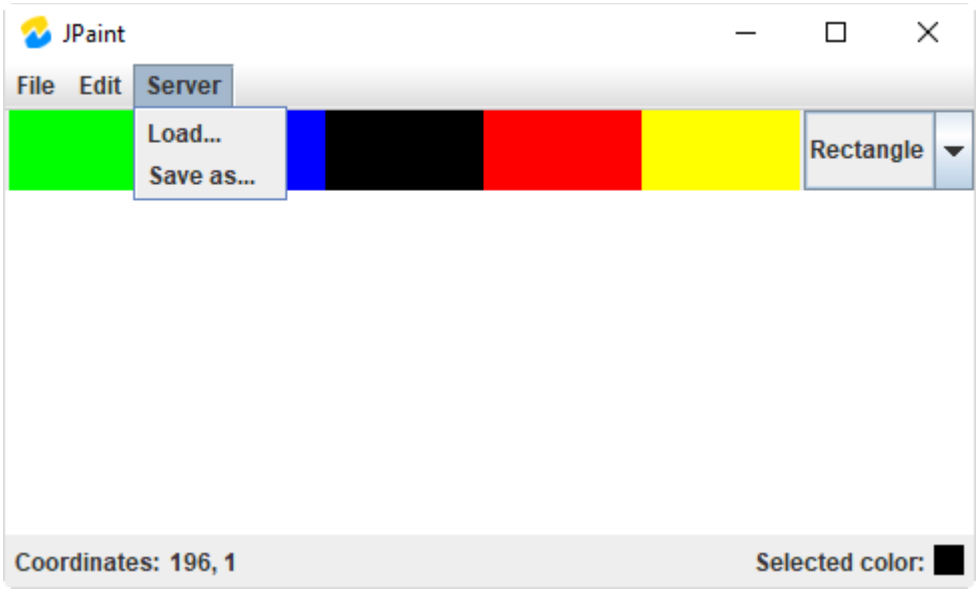
It's okay that all code in `Client` is executed in the main thread. Let it be up to the code that uses `Client` to ensure that the methods called are performed in a background thread. Especially in an application with a graphical user interface. It's also okay that `ClientTest`, which is using `Client`, does not use background threads.

Now you're done, Good work! If you want, you can continue with the optional tasks on the following pages.

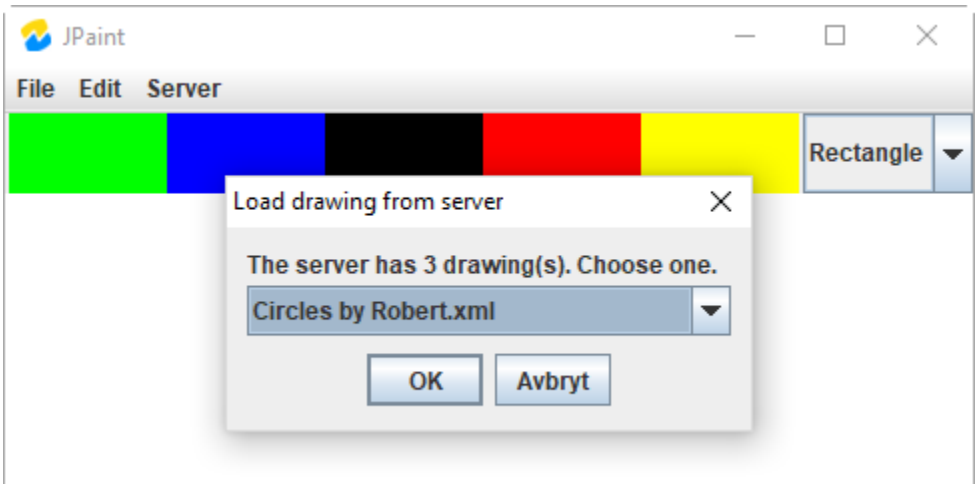
Optional task 1

JFrame

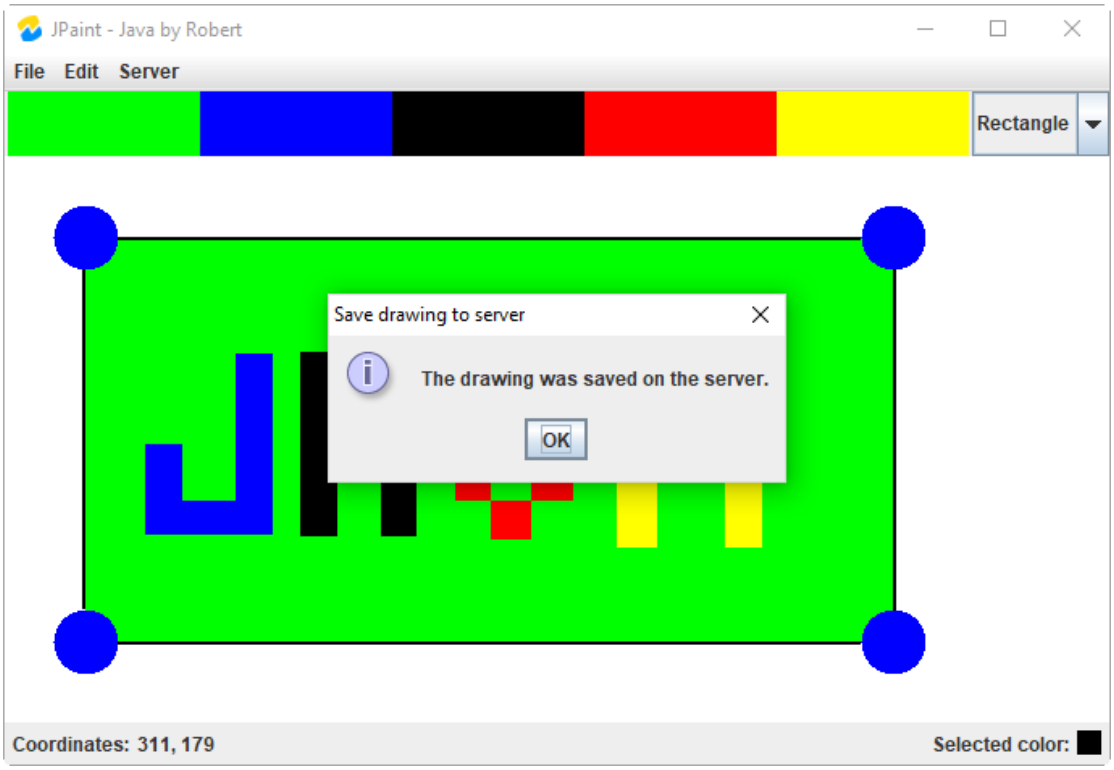
You will rewrite your drawing program so that it can communicate with your server. Your `JFrame` will use the `Client` class to list, load and save files. Start by adding a menu named Server in which menu options Load ... and Save as ... are included.



When the menu option Server | Load ... is selected, the client will connect to the server and request a list of all files the server has (only XML files will be returned by the server). The names of all files from the server will appear in the client as a selectable list in a dialog box (see example in How to make dialogs in The Java Tutorials from Oracle). Once the user has selected a file in the list, this file should be downloaded/copied from the server to the client. Drawing area in the window will be updated to show the downloaded file.



When the menu option Server | Save as ... is selected, the current drawing in the drawing area will be saved to the server with the file name that is typed into the dialog. In the dialog box, a suggested name appears, same as we have in the menu item File | Save as . When the user clicks ok, the current drawing must be saved as a file on the server (it's ok to save the file locally to the hard disk first).



Remember that the communication to/from the server should NOT be done on the event dispatch thread because the communication will be classified as a long-running tasks that can block other events occurring in the program (like mouse clicks). Use a `SwingWorker` for each "command" that is performed against the server.

Assignment8

This class is attached with this description. The class is used to create and display your `JFrame`. You can change the code so that the correct name of the class is used. Start by changing the `main`-method to check if there is one or more arguments (`args.length`). If so, use the first argument as the address of the server. You can use another argument as the port that server is listening to.

If no argument is given, or if the second argument cannot be converted to a number, the client should use the address "localhost" or "127.0.0.1" (which works best in your system) and the port should be 10000 (the port may change in another assignment).

Optional exta task 2

You will create executable jar files for your solution. One jar file to start the server and another jar file to start the client (your JFrame will be displayed). Name these files to server.jar and client.jar respectively. Include these in your zip file that you will submit.