CS 251 Outlab 8: Java

Please refer to the general instructions and submission guidelines at the end of this document before submitting. 100% penalty on all group members for mistakes in submission format.

Task 1: Socket Programming (20)

Create a simple in-memory key-value store as a Java based client-server application using socket programming. Before you begin, familiarize yourself with the basics of socket programming.
The server stores the key-value pairs in an in-memory database (ie. non-persistent database -- a map would be a good choice). Keys and values are both integers, values being the "weight" of the key.
The client may request server to add a key, increase its weight (by one) in the database, or read the weight of a key.

BasicServer (10)

Create a class name **BasicServer** in file **BasicServer.java** that behaves as follows:

- Spawns a socket and listens ("binds") on port **5000** for connection from clients.
- After a client connects, the server reads request from a client, executes it, writes a response to the client and a message (1 line per message) on the console.
- The server terminates only when explicitly killed with SIGINT (Ctrl+C).

Remember, the server stores frequencies of keys.

Command (recvd from client)	Server Action	Response to Client	Console Output
N.A.	Create a socket	N.A.	`Listening on <port no>`
`add <int>`	Increment weight of key	Weight of key (post increment)	`ADD <int>`
`read <int>`	Fetch weight of key	Weight of key=<int>, 0 if key missing	`READ <int> <int>` Key Weight
`disconnect`	Close connection to client	N.A.	`DIS`

BasicClient (10)

Create a class name **BasicClient in** file **BasicClient.java** that behaves accordingly. It reads commands from a file (the only argument to the client program). The commands are as follows:

1. `connect <server-ip> <server-port>`
   Opens a socket to connect at specified IP address and port no. This starts a "session".
   "**OK**" should be displayed for a successful connection, "**No Server**" otherwise and quit (without printing sum of deltas).
   This will be the **first command** in the file (but **may appear many times** in the input file).
   All subsequent commands must succeed only if there is an active connection to the server.
2. `add <int>`
   Send the `add <int>` command to server, store the server's response. You'll need it for the next command.
3. `read <int>`

Print “`<weight> <delta>`”, where weight is the server’s response and delta is the difference between the last known weight of this key (0 being the default weight) and the current weight. Delta can be a 0 or positive.

4. `disconnect`
   Send the `disconnect` command to the server and immediately terminate current connection to the server. “**OK”** should be displayed for a successful disconnection, which marks the end of a session. (The deltas should not be reset).
5. `sleep <int>`
   Client must sleep for specified number of **milliseconds**. This command will only occur immediately after a `disconnect` (but it may not necessarily appear).

After all commands are processed, the client must print the sum of all deltas in all the sessions and terminate.

NOTE: We will not be testing exception handling (many, many things can go wrong when you have sockets). That being said, your programs should run within our reasonable expectations. (For ex, we will run the client only after the server is ready, two clients won’t contend for the server at the same time, etc. -- all that is in task 2.)

Task 2: Multi-threading (20)

MultiServer

While a client can connect to one server at a time, a server can be concurrently connected to multiple clients. Extend KV store server from Task 1 to do the same. Create a class name **MultiServer** in file **MultiServer.java**, extends the behaviour of **BasicServer**:

1. The server should maintain a common KV store/DB across all clients and all server threads.
2. The server should accept a client’s connection and delegate processing of the client to a new thread. The thread should terminate after client disconnects.

You may use ThreadPools or any such mechanisms for this. You are forbidden from using any external libraries.
NOTE: Solve the synchronization problem for the multithreaded database.
NOTE: No change in client is required. Task 1 `BasicClient` should work with `MultiServer`.
NOTE: The client exits when either “No Server” is there or all the commands in the file have been processed. However, client may reconnect multiple times.

Task 3: Pokemon <3 (50)

*This task will help you understand the essence of **multiple implementations of a single interface**.*
You are given some skeleton code for this question, please fill in the missing parts.

`FetchAndProcess.java` contains an Interface **`FetchAndProcess`** which declares two functions namely `fetch()` and `process()`. Observe that there is no requirement for function implementations in an interface, just signatures are enough sometimes.

`FetchAndProcessFromDisk`, `FetchAndProcessFromNetwork` and **`FetchAndProcessFromNetworkFast`** each “implement” **`FetchAndProcess`** interface. Hence, all these classes implement the functions that the interface **`FetchAndProcess`** “provides”.

You **do not** need to change anything in `Main.java`. You only need to make changes in other Java files. The implementations given to you are currently empty which you need to fill.

You can run the implementation using:

```
$ javac -cp sqlite-jdbc-3.27.2.1.jar:. Main.java
$ java -cp sqlite-jdbc-3.27.2.1.jar:. Main <impl-option>
```

where `<impl-option>` can be one of `{disk, net, fastnet}` depending on what implementation you want to run. The JAR file provides the SQLite driver.

Sub tasks:

1. Fetch pokemon names from disk, store them in a sqlite database called `pokemon.db`.
2. Fetch pokemon names from the internet, add them to the same database.
3. Report some facts.

Disk (10)

Modify ***FetchAndProcessFromDisk.java***
The **fetch()** function has the argument **List<String> paths**. When you run **javac Main.java && java Main disk**, a list of file paths (of files in data dir) will be passed as **paths** parameter into `FetchAndProcessFromDisk` class `fetch()` method.

Each file has one pokemon name per line. The number of lines in a file can vary.
The **fetch()** should **for each file path**

1. Open the file and read from it the names.
2. You have to associate the filenames (**only** the filename and **not** the directory stem, and **no** extension) to pokemon names for the `process()` method using a `Map<String, String>` (since that is used by `process()`).

Once `fetch()` is done the field ~~**List<String> data**~~ `Map<String, String>` should have names of **all pokemon** from **all paths**. Pokemon names are NOT unique across all paths.

In Main.java just after fetch(), process() is called. We describe process() later.

Network (10+5)

Modify ***FetchAndProcessFromNetwork.java, FetchAndProcessFromNetworkFast.java***
This task is similar to the previous one, with the difference being that paths passed to `FetchAndProcessFromNetwork` class fetch() method are now urls instead of file paths. You need to connect to URLs instead of opening files and get the data from them. The format of output you get when you connect to a url is exactly the same as that described in previous task.

The **fetch()** should **for each url path**

1. Open the URL and read from it. See this for hints.
2. You have to associate the URLs to pokemon names for the `process()` method using a `Map<String, String>` (since that is used by `process()`).

Once fetch() is done the field ~~**List<String> data**~~ `Map<String, String>` should have names of **all pokemon** from **all paths**. Pokemon names are NOT unique across all paths.

You should connect to each URL sequentially. This may take some time as each url connection takes some time.

An additional 5 marks will be awarded if you fetch the URLs in parallel (**FetchAndProcessFromNetworkFast.java)**. You may use any technique for parallelization. Note that, we will compare the speed of your sequential (**FetchAndProcessFromNetwork.java)** and parallel implementation (don't just submit the parallel)

Process -- storing in a DB (25)

The `fetch()` just declares the signature. **You must add a default implementation of the process() method in the interface class.** This method is responsible for storing the fetched data into a SQLite database (`pokemon.db`).
Please understand what the above means! All the database code goes in the `FetchAndProcess.process()`. You cannot add `FetchAndProcessFromDisk.process()`!
This method must open a connection to the database, create the following table ONLY IF NECESSARY and insert the data. Make use of the JDBC interface for managing connection with the database. SQLite specific details are here.
Tablename: pokemon
Columns:

- `pokemon_name`,
- `source_path` (URL/filename)

The DB should not contain any duplicate rows after your programs are run. The best way to ensure this is by using a uniqueness constraint on the database. You'll **have to handle failed** (duplicate) insertion attempts.

Additionally, for the network fetch classes, the `process()` method does one additional step. It reports (ie. prints them one per line) the pokemons that were found in more than one `source_path`. This cannot be determined from just data fetched from the network (since the DB may not be empty at this point!). (**5 points**)
Hint: Make the DB compute this set!

Note:

1. Your code will be tested on hidden test cases also.
2. You cannot assume that the DB is necessarily empty (but it will have unique entries if it is not empty) when your code is run, and any ordering of running the programs.

General Instructions

- Make sure you know what you write, you might be asked to explain your code at a later point in time
- The submission will be graded automatically, so stick to the naming conventions strictly
- The deadline for this lab is **Sunday, 13th October, 23:55.**

Submission Instructions

After creating your directory, package it into a tarball **<team_name>-outlab8.tar.gz.** Submit only once per team from the moodle account of the smallest roll number. You may use the following command:

```
$ tar czf <team_name>-outlab8.tar.gz path/to/<team_name>-outlab8/
```

We will untar your submissions using

```
$ tar xzf <team_name>-outlab8.tar.gz
```

Make sure that when the above is executed, the resulting <team_name>-outlab8/ directory has the correct directory structure.

The directory structure should be as follows (nothing more, nothing less).

<team_name>-outlab8

```
├──── BasicServer.java
├──── BasicClient.java
├──── MultiServer.java
├──── FetchAndProcessFromDisk.java
├──── FetchAndProcessFromNetwork.java
├──── FetchAndProcessFromNetworkFast.java
├──── FetchAndProcess.java
└──── references.txt(optional)
```