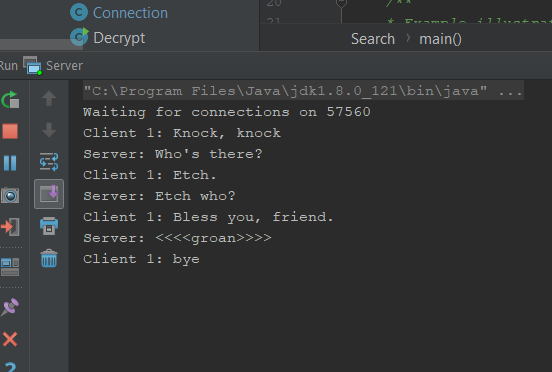
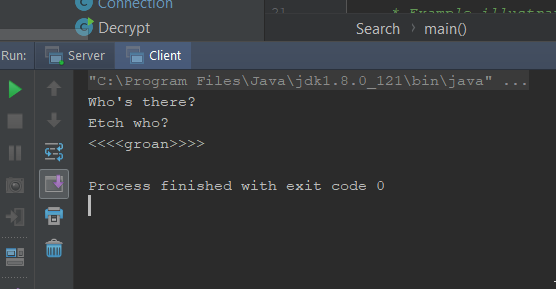
**Task 1:**

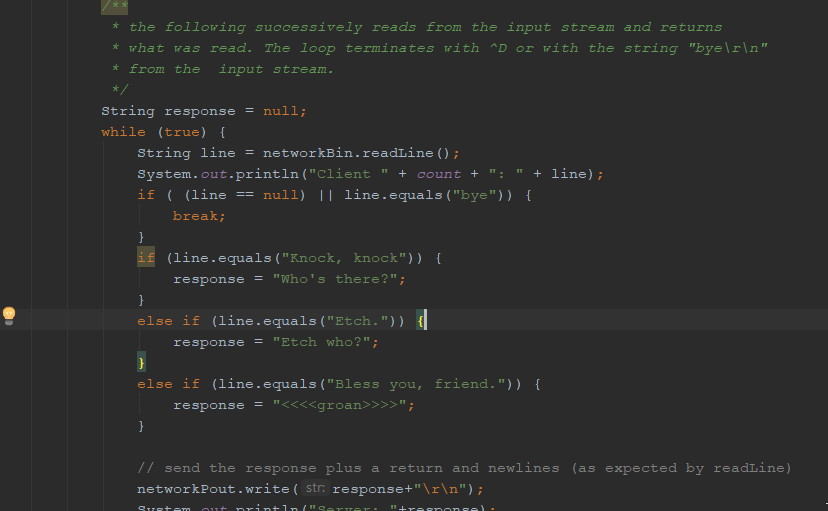
Server output with updated knock-knock joke:

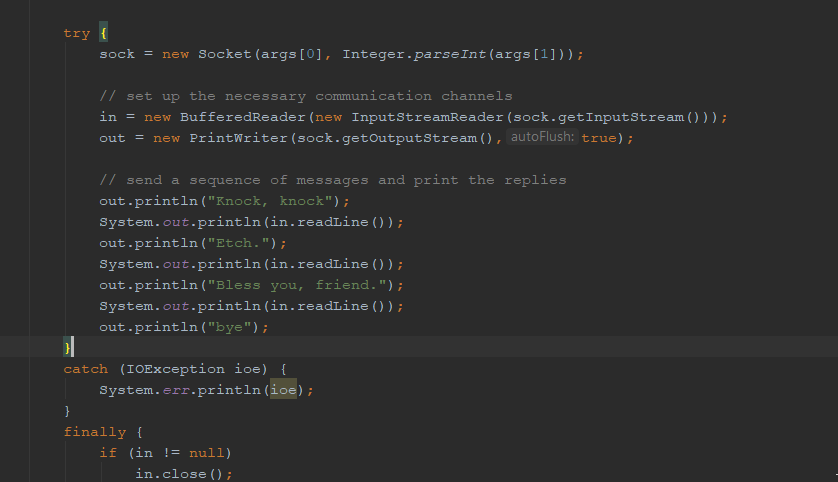


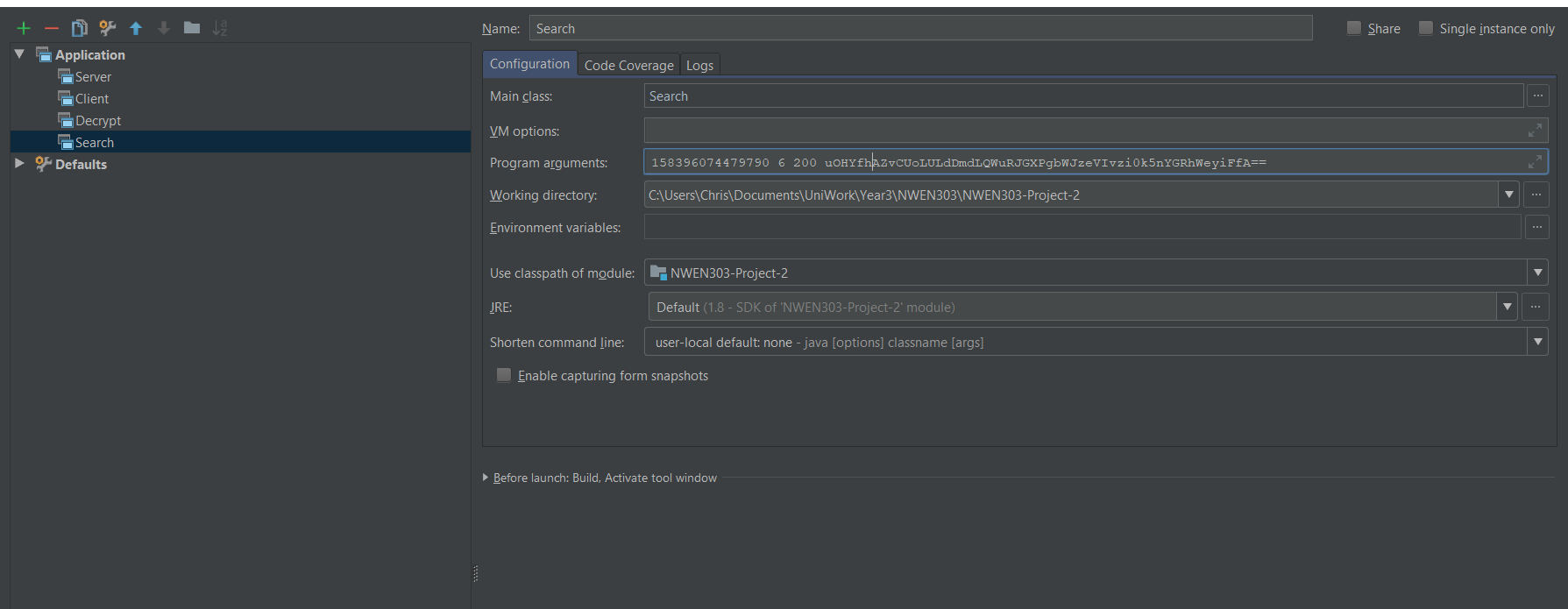
Client output with updated knock-knock joke:



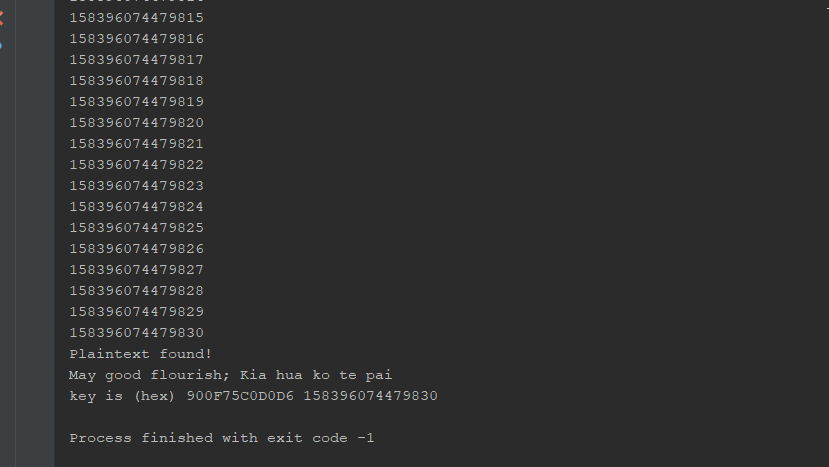
To do this I had to edit the Connection.java file that handles the connections between client and server and the EchoClient.java to make messages passed between the two concordant.





Second part: Modified Search.java to include key range parameter (3rd argument of 200):

I added: **int keyRange = Integer.parseInt(args[2]);** to the main method in Search.java and edited the for loop to iterate for keyRange instead of 99. The output is still the same, as it still finds the key with a key range of 99. E.G as shown here:



**Task 2:**

1. Control flow outline

Setup:

A KeyManager is created with an initial key, keysize and ciphertext.

A client is run which opens a new socket and the KeyManager accepts this client and creates a connection (called KeyManagerConnection). If this is the first connection, the KeyManager starts a timer.

Begin:

Before requesting work, our client needs to try send a message to the server (KeyManager socket). We shouldn’t do anything if it’s not running, hence why we try to send this message.

Next, we send a message seeing if there is work still to be done.

The format for this message is: ‘Work\_Left?’

The server responds back with message containing a true or false value. If this value is false, our client will keep sending this message checking if there is working to be done. When there is no work to be done, the server closes all connections to it, so it makes these connections for a limited time until its connection to the server is forcibly broken.

Given there is work left,

A client sends a work request for chunksize keys to KeyManagerConnection and waits for a response.

Format for this message: ‘Requesting Work: <ChunkSize>’

The KeyManager recognizes this message and does some work to determine what work to give the client. It sends back a message to the client with the required information to do work. As we are assured that the client will complete the work, the server also changes the ‘currentKey’ value so that if another client requests work, it will be given different work.

Format for this message:

*‘InitialKey: <InitialKey>\tChunkSize: <ChunkSize>\tKeySize: <KeySize>\tCipherText: <CipherText>’*

* Note: If the chunksize requested by the client is larger than the tasks left in the key manager, we just send back a chunk size of the remaining tasks so that client gets all the tasks.

Our client now has a message back from the KeyManager containing the information required to do work. The connection is closed at this point.

The client then works through the key space, trying each key and checking if it matches the expected decrypted plaintext.

If no key is found, we create a connection to a KeyManagerConnection and send a message informing it that with our work we couldn’t find it with message ‘No Key Found’. This is registered but not responded to and the client closes the connection immediately. This process of requesting, retrieving and sending results are then repeated until some client finds the key or we exhaust the key space.

If the key is found, we create a connection to a KeyManagerConnection and send a message informing it of this so no more work is requested and responded to by other clients. Once the key has been found, the time taken is output, the sockets are closed and thus all connections by clients are closed. Hence, nominal end to program.

1. How this design satisfies each of the system requirements

* Clients only need to be aware of the location of the key manager
  + The key manager creates a socket that is open for clients to connect to. In this way, the clients are only aware of their connection to the Key Manager via their socket connection in KeyManagerConnection.
* Clients can join or leave but will complete the work they have been requested
  + Clients can easily join by running ‘Java Client hostname port chunksize’. Any number of clients can join by just running different instances of a terminal and running this command, specifying the hostname and port that the KeyManager socket is open on.
  + In terms of leaving, it’s a bit more complicated. The control flow design between the Client and socket connection doesn’t stop the client from being able to leave. What the design does do is make it easy to identify periods of execution where the clients can and cannot leave. So, for instance when requesting work or doing work to be sent back to the KeyManager, we cannot leave. Thus, when not inside these execution paths, we are permitted to leave.
  + In reality, a client will try leave by running an interrupt (like Ctrl-C) on the running program in the terminal. By creating a custom hook to wait until we’ve completed our work before exiting, clients can leave, but only when the work has been completed.
* Clients request work from the key manager and return results to it.
  + The key manager has an open socket that accepts connections from Clients. A client sends a message as specified in the control flow for ‘Requesting Work’. It also sends another message when work has completed for the result of the work too. This socket-based message passing design allows for this requirement to be satisfied.
* Connections between clients and the masters only exist long enough to request work or to return results
  + To satisfy this, we only open and close connections long enough to get what we want or send what we want. Socket connections need to be manually closed, so this design doesn’t automatically do this upon completion of a retrieval or post of information to the key manager, in each instance of communication, we know when we request work it’s a simple control flow of:

Open connection -> request work -> get work -> close connection

Or when we send a message not expecting a response it’s just:

Open connection -> send message -> close connection.

* When the key is found, the key manager will shutdown
  + When a client finds the key, a message is sent to the key manager. The key manager recognizes this message and shuts down the socket connection and itself, hence closing all connections to it and shuts itself down. Again, message passing design allows this.

**Task 3:**

1. Code implemented. See files KeyManager.java, Client.java, KeyManagerConnection.java,