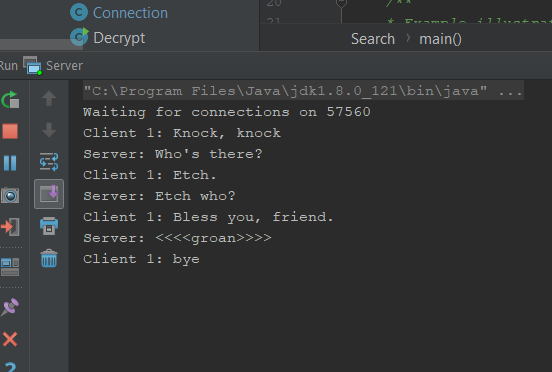
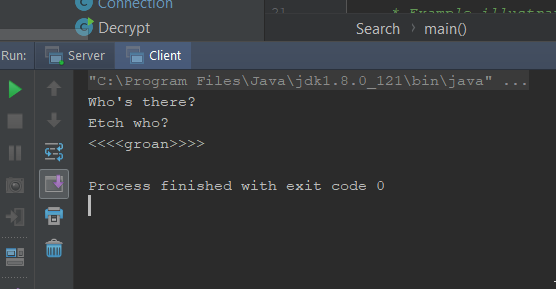
**Task 1:**

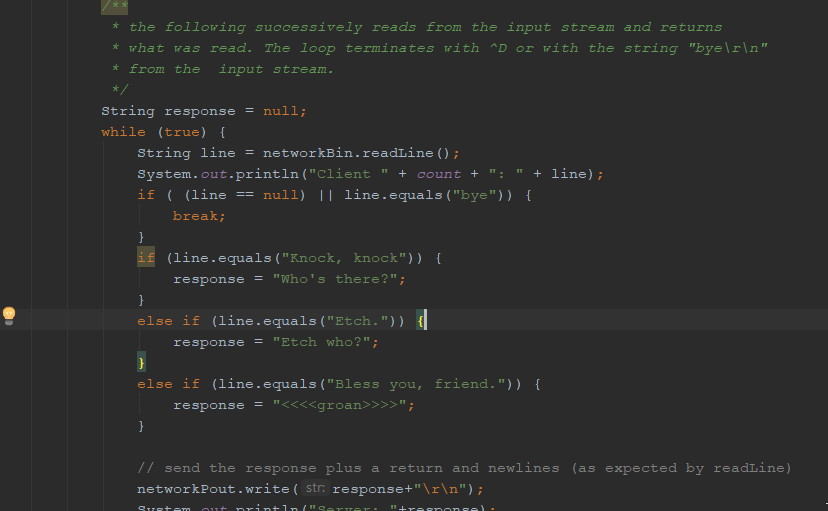
Server output with updated knock-knock joke:

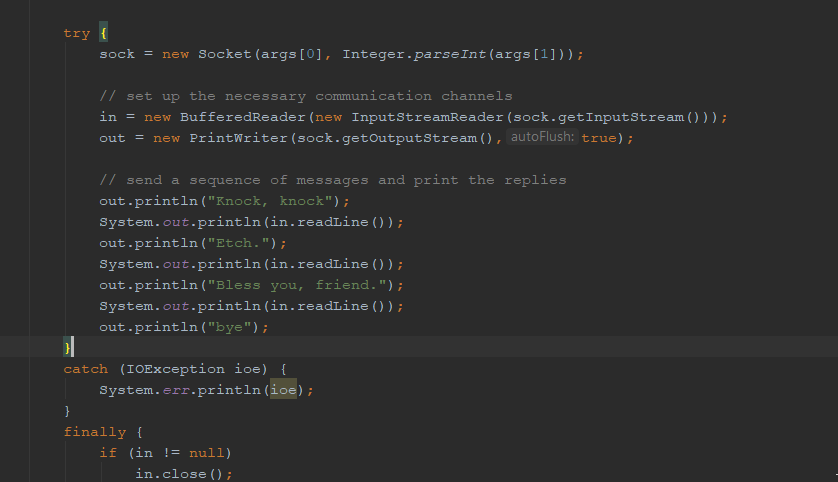


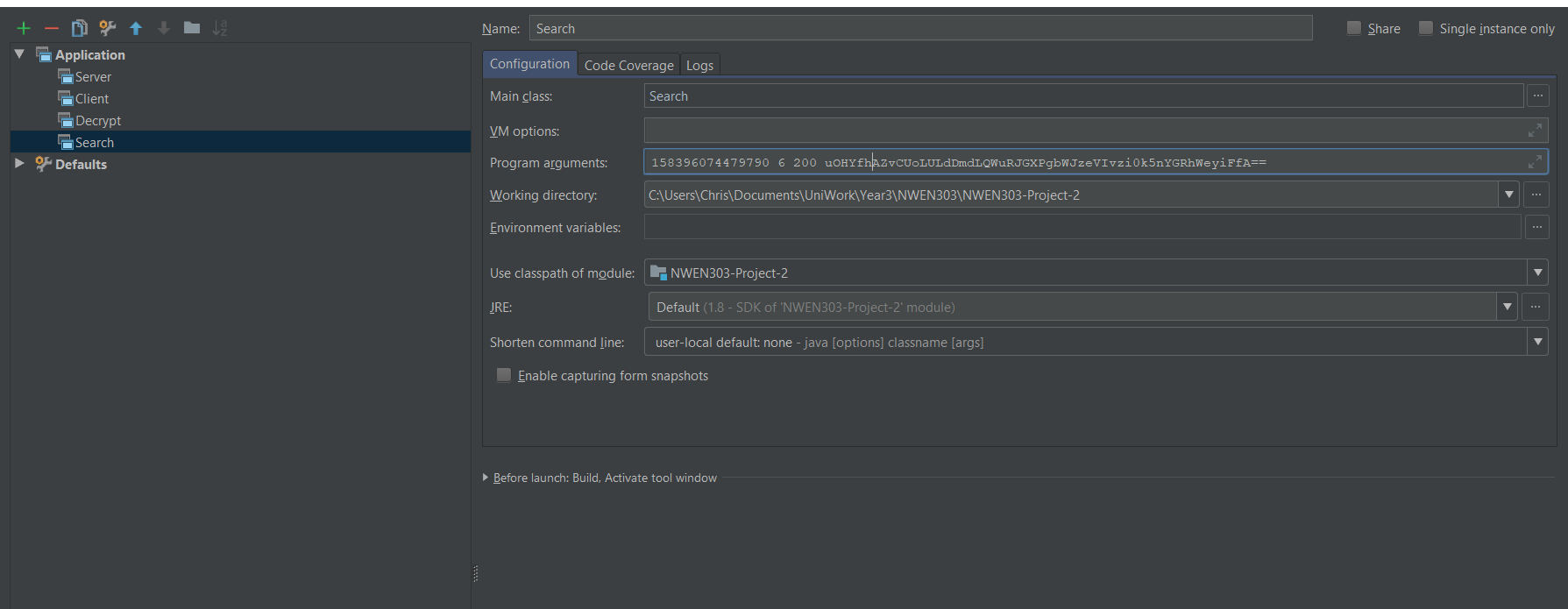
Client output with updated knock-knock joke:



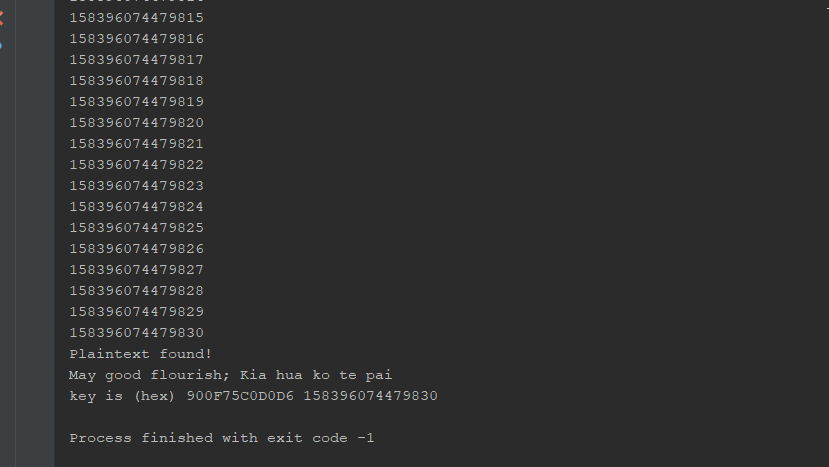
To do this I had to edit the Connection.java file that handles the connections between client and server and the EchoClient.java to make messages passed between the two concordant.





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 opens a new socket and the KeyManager accepts this client and creates a connection. If this is the first connection, the KeyManager starts a timer.

Begin:

A client sends a work request for chunksize keys to KeyManager and waits for a response specifying the startKey and the chunksize it will work through.

The KeyManager allocates a start key and key space (usually the chunk size the client wants) based upon it’s bag of tasks and returns it back to the client.

* 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.
* If the tasks left in the key manager is 0, we close the connection to the client and don’t pass a message back.
* Tasks are removed from the bag when sent to a client so that we don’t have multiple clients working on the same datasets.

Our client now has a message back from the KeyManager containing the startKey and chunksize to work through. The connection is closed at this point.

The client then works through the key space – work begins and then work ends.

The client then creates a connection to the KeyManager to send a map of {key -> plaintext decryption result} back to the key manager. Connection closes after message is sent.

The key manager checks these map values against the expected plaintext.

If any of the values match the expected plaintext then we have found a key match. If so, we close all our connections to our clients and print out the key and plaintext decryption result. We also stop our timer and print the time taken to find our key too. We then shutdown our key manager and all client connections.

As the connection has been closed to the client (after sending the results), the client is free to start its loop again (from ‘*Begin:*’ above) and begin requesting tasks. However, if the KeyManager finds the key while there’s a current connection to a client, the work done on that client will be lost as when the KeyManager closes, are client connections are disconnected.

1. How this design satisfies each of the system requirements

* Clients only need to be aware of the location of the key manager
  + The client will only need to be aware of the specific memory address location of the Key Manager because