Exercise 02

Name: Jose Juan Sandoval

Link to Project: https://github.com/Juanchiselo/CS380/tree/master/Exercises/Exercise%2002

Java Code

Ex2Client.java

```java
package Exercise02;

import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;

public class Ex2Client
{
    private static Socket socket;

    public static void main(String[] args)
    {
        connectToServer();
    }

    /**
     * Connects the client to the server and
     * creates a Listener thread.
     */
    public static void connectToServer()
    {
        String hostName = "codebank.xyz";
        int portNumber = 38102;

        try
        {
            socket = new Socket(hostName, portNumber);
            new ListenerThread(socket).start();
            System.out.println("Connected to server.");
        }
        catch (UnknownHostException e)
        {
            System.err.println("ERROR: Unknown host " + hostName + ".");
        }
        catch (Exception e)
        {
            System.err.println("ERROR: Could not connect to " + hostName + ".");
        }
    }

    /**
     * Disconnects the client from the server.
     */
    public static void disconnectFromServer()
    {
        try
        {
            socket.close();
            System.out.println("Disconnected from server.");
        }
        catch (IOException e)
        {
            System.err.println("ERROR: " + e.getMessage());
        }
    }
}
```

ListenerThread.java

```
package Exercise02;

import java.io.*;
import java.net.Socket;
import java.nio.ByteBuffer;
import java.util.zip.CRC32;
import java.util.zip.Checksum;

public class ListenerThread extends Thread
{
    public volatile static boolean endThread = false;
    private Socket socket = null;

    public ListenerThread(Socket socket)
    {
        super("Listener Thread");
        this.socket = socket;
    }

    /**
     * The overridden run() function belonging to the Thread class.
     * This is what handles the communication between the server and the client.
     */
    public void run()
    {
        try
        {
            // Variable and constant to track
            // the amount of server's responses.
            int counter = 1;
            final short BYTES_TO_RECEIVE = 100;

            // Variables and array to hold the server's responses.
            int firstNibble;
            int secondNibble;
            int reconstructedByte;
            byte receivedBytes[] = new byte[BYTES_TO_RECEIVE];
            String receivedBytesString = "  ";

            // Constants for bitwise operations.
            final short NIBBLE_SIZE = 4;
            final int BITMASK = 0xFF;

            // InputStream object needed for receiving
            // and reading the server's responses.
            InputStream inputStream = socket.getInputStream();

            // The main loop of execution.
            // InputStream.read() returns -1 when the end of the stream
            // has been reached and there was no byte to be read.
            while((firstNibble = inputStream.read()) != -1
                    && (secondNibble = inputStream.read()) != -1)
            {
                // The byte obtained from combining the first and second nibbles sent by the
server.
                // The first nibble gets LEFT SHIFTED by the size of a nibble, meaning 4 bits.
                // The second nibble gets ANDed with the 0xFF bitmask.
                // ANDing the second nibble with the bitmask is necessary because some Java
                // architectures do integer promotions and not using the bitmask may yield wrong
results.
                // Finally both nibbles are ORed with each other to combine them into a single
byte.
                reconstructedByte = (firstNibble << NIBBLE_SIZE) | (secondNibble & BITMASK);

                receivedBytesString += Integer.toHexString(reconstructedByte).toUpperCase();
                receivedBytes[counter - 1] = (byte) reconstructedByte;

                // This only separates the received bytes
                // into groups of 20 for user convenience.
```

```
            if(counter % 10 == 0
                    && counter != BYTES_TO_RECEIVE)
                receivedBytesString += "\n  ";
            else if(counter == BYTES_TO_RECEIVE)
            {
                System.out.println("Received bytes:\n" + receivedBytesString);
                long checksum = verifyData(receivedBytes);
                System.out.println("Generated CRC32: "
                        + Long.toHexString(checksum).toUpperCase() + ".");
                respondToServer(checksum);
                break;
            }

            counter++;
        }

        Ex2Client.disconnectFromServer();
    }
    catch (IOException e)
    {
        System.err.println("ERROR: Connection lost with server.");
    }
    catch (Exception e)
    {
        System.err.println("ERROR: " + e.getMessage());
    }
}

/**
 * Verifies the integrity of the received data
 * by calculating its CRC32 checksum.
 * @param data - The data to calculate the checksum for.
 * @return - Returns the checksum.
 */
private long verifyData(byte[] data)
{
    Checksum checksum = new CRC32();
    checksum.update(data, 0, data.length);
    return checksum.getValue();
}

/**
 * Breaks up the checksum into a sequence
 * of 4 bytes to send to the server.
 * @param checksum - The checksum to be broken into 4 bytes.
 * @return - A byte array filled with the 4 bytes.
 */
private byte[] prepareResponse(long checksum)
{
    // Allocates 8 spaces in the ByteBuffer.
    // Long.BYTES is the same as Long.SIZE/Byte.SIZE which equals 64/8.
    // A long is 64 bits and a byte is 8 bits.
    ByteBuffer buffer = ByteBuffer.allocate(Long.BYTES);

    // Writes 8 bytes into the ByteBuffer
    // containing the given long value.
    buffer.putLong(checksum);

    // Gets the array that backs the ByteBuffer.
    byte checksumReturn[] = buffer.array();

    // A new array to hold the response because
    // we only need to send 4 bytes back to the server.
    // NOTE: The first 4 bytes of the ByteBuffer array
    // are filled with zeroes so we just need the last 4.
    // TODO: Find a way to remove this step.
    byte response[] = new byte[4];
    for(int i = 0; i < 4; i++)
        response[i] = checksumReturn[i + 4];

    return response;
```

```
    }

    /**
     * Responds to the server with the 4 byte sequence
     * obtained from the given checksum.
     * @param checksum - The checksum to be sent to the server.
     */
    private void respondToServer(long checksum)
    {
        try
        {
            socket.getOutputStream().write(prepareResponse(checksum));

            int serverResponse;
            if((serverResponse = socket.getInputStream().read()) == 1)
                System.out.println("Response good.");
            else
                System.out.println("Bad response. Server returned " + serverResponse);
        }
        catch (IOException e)
        {
            System.err.println("ERROR: " + e.getMessage());
        }
    }
}
```