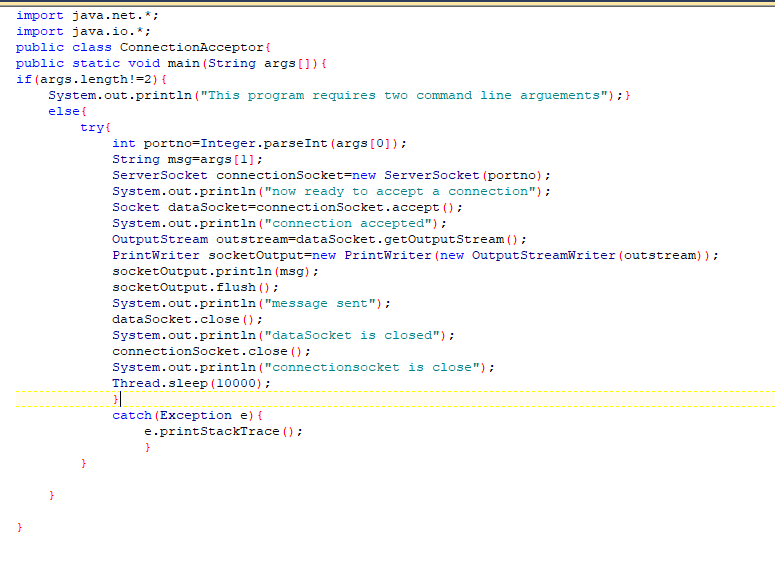
## LAB NO:02

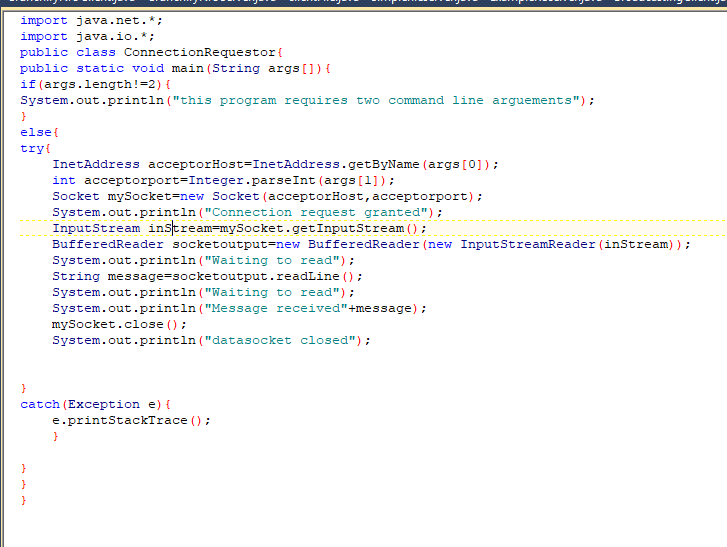
## OBJECTIVE:- To get familiar with stream sockets.

**Task # 01:-Compile and run the above code. Start the acceptor first and then the requestor with appropriate command line arguments. Describe and explain the output.**

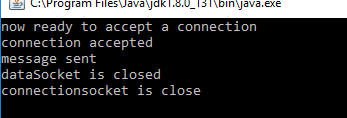
**Server class:**

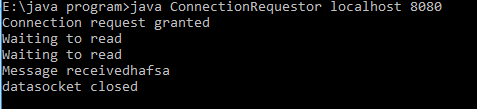


**Client class:**





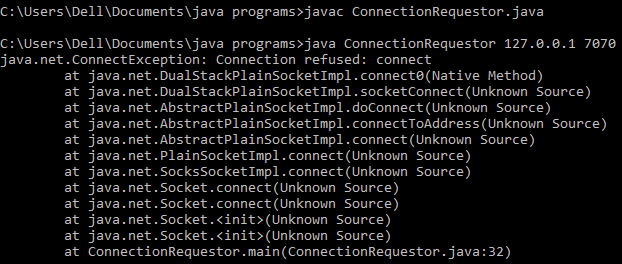




**Task # 02:- Compile and run the above code, but reverse the order of program’s execution. Start the requestor first and then the acceptor. Describe and explain the outcome.**

**OUTPUT:-**

If we run client program first then it will gives you exception because connection is not established.



**Task #03:- Add a time delay of 5 seconds in the ConnectionAcceptor process just before the message is written to the socket, then run the program. This will show you the blocking at the receiver. Show a trace of the output of the processes.**

**//Client class:**

import java.net.\*;

import java.io.\*;

//this application requests a connection and sends a message

// using the stream mode socket.

public class ConnectionRequestor {

public static void main(String[] args){

if(args.length!=2){

System.out.println("This program requires two command line arguments");

// the arguments are

//host name of connection acceptor and port number of connection acceptor

}

else{

try{

InetAddress acceptorHost=InetAddress.getByName(args[0]);

int acceptorPort=Integer.parseInt(args[1]);

Socket mySocket=new Socket(acceptorHost,acceptorPort);

System.out.println("Connection request granted.");

InputStream inStream=mySocket.getInputStream();

//create buffered reader object for character mode output

BufferedReader socketInput=new BufferedReader(new InputStreamReader(inStream));

System.out.println("Waiting to read.");

String message=socketInput.readLine();

System.out.println("Message received."+"\t"+message);

mySocket.close();

System.out.println("data socket closed.");

Thread.sleep(10000);

}catch(Exception ex){

ex.printStackTrace();}

}}}

**SERVER CLASS**

import java.net.\*;

import java.io.\*;

public class ConnectionAcceptor {

//Two command line arguments are needed

// port number of the server socket and second is the message to send

public static void main(String[] args){

if(args.length!=2){

System.out.println("This program requires two command line arguments");

}else{

try{

int portNo=Integer.parseInt(args[0]);

String message=args[1];

ServerSocket connectionSocket=new ServerSocket(portNo);

System.out.println("now ready to accept a connection");

Socket dataSocket=connectionSocket.accept();

System.out.println("Connection Accepted");

OutputStream outStream=dataSocket.getOutputStream();

//create a print writer for character mode output

PrintWriter socketOutput=new PrintWriter(new OutputStreamWriter(outStream));

//write a message into the data stream

**Thread.sleep(5000); //here we make some changes in program**

socketOutput.println(message);

//the ensuing flush method ensures that data is written into the data socket before the socket is closed.

socketOutput.flush();

System.out.println("message sent");

dataSocket.close();

System.out.println("data socket closed.");

connectionSocket.close();

System.out.println("connection socket closed.");

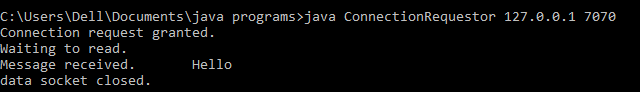
Thread.sleep(10000);

}catch(Exception ex){ ex.printStackTrace();

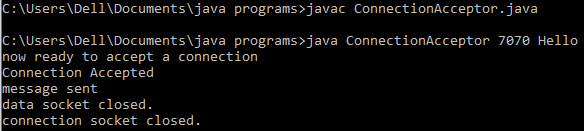
} } } }

**NO CHANGE IN THE OUTPUT ONLY CHANGE IN THEIR TIMING.**

**OUTPUT OF SERVER**



**OUTPUT OF CLIENT:-**



**Task #04:-** **Modify the sample code to include two way communication between the client and the server.**

**Program for ChatServer:-**

**import java.io.\*;**

**import java.net.\*;**

**import java.util.\*;**

*// This is the chat server program. Press Ctrl + C to terminate the program.*

**public class ChatServer {**

private int port;

private Set<String> userNames = new HashSet<>();

private Set<UserThread> userThreads = new HashSet<>();

**public ChatServer(int port) {**

this.port = port; **}**

**public void execute() {**

**try (ServerSocket serverSocket = new ServerSocket(port)) {**

System.out.println("Chat Server is listening on port " + port);

**while (true) {**

**Socket socket = serverSocket.accept();**

System.out.println("New user connected");

**UserThread newUser = new UserThread(socket, this);**

userThreads.add(newUser);

newUser.start();**}**

**}** catch (IOException ex) **{**

System.out.println("Error in the server: " + ex.getMessage());

ex.printStackTrace();**}}**

**public static void main(String[] args) {**

if (args.length < 1) **{**

System.out.println("Syntax: java ChatServer <port-number>");

System.exit(0);}

int port = Integer.parseInt(args[0]);

**ChatServer server = new ChatServer(port);**

server.execute();**}**

*//Delivers a message from one user to others (broadcasting*)

**void broadcast(String message, UserThread excludeUser) {**

for (UserThread aUser : userThreads) **{**

if (aUser != excludeUser) **{**

aUser.sendMessage(message);

**}}}**

*// Stores username of the newly connected client.*

**void addUserName(String userName) {**

userNames.add(userName);**}**

*//When a client is disconneted, removes the associated username and UserThread*

**void removeUser(String userName, UserThread aUser) {**

boolean removed = userNames.remove(userName);

if (removed) **{**

userThreads.remove(aUser);

System.out.println("The user " + userName + " quitted");**}}**

**Set<String> getUserNames() {**

return this.userNames; **}**

*// Returns true if there are other users connected (not count the currently connected user)*

**boolean hasUsers() {**

return !this.userNames.isEmpty();**}}**

**Program for Chat Client:-**

import java.net.\*;

import java.io.\*;

// This is the chat client program. Type 'bye' to terminte the program.

**public class ChatClient {**

private String hostname;

private int port;

private String userName;

**public ChatClient(String hostname, int port) {**

this.hostname = hostname;

this.port = port; **}**

**public void execute() {**

**try {**

**Socket socket = new Socket(hostname, port);**

System.out.println("Connected to the chat server");

**new ReadThread**(socket, this).start();

**new WriteThread**(socket, this).start();

**}** catch (UnknownHostException ex) **{**

System.out.println("Server not found: " + ex.getMessage());

**}** catch (IOException ex) **{**

System.out.println("I/O Error: " + ex.getMessage());}}

**void setUserName(String userName) {**

this.userName = userName; **}**

**String getUserName() {**

return this.userName**; }**

**public static void main(String[] args) {**

if (args.length < 2) return;

String hostname = args[0];

int port = Integer.parseInt(args[1]);

**ChatClient client = new ChatClient(hostname, port);**

client.execute(); **}}**

**Program for ReadThread class:-**

import java.io.\*; import java.net.\*;

*//This thread is responsible for reading server's input and printing it to the console.*

*//It runs in an infinite loop until the client disconnects from the server.*

**public class ReadThread extends Thread {**

private BufferedReader reader; private Socket socket;

private ChatClient client;

**public ReadThread(Socket socket, ChatClient client) {**

this.socket = socket; this.client = client;

**try {**

**InputStream input = socket.getInputStream();**

reader = new BufferedReader(new InputStreamReader(input));

**}** catch (IOException ex) **{**

System.out.println("Error getting input stream: " + ex.getMessage());

ex.printStackTrace(); **}}**

**public void run() {**

while (true) **{**

**try {**

String response = reader.readLine();

System.out.println("\n" + response);

*// prints the username after displaying the server's message*

if (client.getUserName() != null) **{**

System.out.print("[" + client.getUserName() + "]: ");**}**

**}** catch (IOException ex) **{**

System.out.println("Error reading from server: " + ex.getMessage());ex.printStackTrace();

Break ; **}}}}**

**Program for Write Thread class:-**

import java.io.\*;

import java.net.\*;

*// This thread is responsible for reading user's input and send it to the server.*

*// It runs in an infinite loop until the user types 'bye' to quit.*

**public class WriteThread extends Thread {**

private PrintWriter writer;

private Socket socket;

private ChatClient client;

**public WriteThread(Socket socket, ChatClient client) {**

this.socket = socket;

this.client = client;

**try {**

OutputStream output = socket.getOutputStream();

writer = new PrintWriter(output, true);

**} catch (IOException ex) {**

System.out.println("Error getting output stream: " + ex.getMessage());

**ex.printStackTrace();}}**

**public void run() {**

Console console = System.console();

String userName = console.readLine("\nEnter your name: ");

client.setUserName(userName);

writer.println(userName);

String text;

**do {**

text = console.readLine("[" + userName + "]: ");

writer.println(text);

**} while (!text.equals("bye"));**

**try {**

socket.close();

**} catch (IOException ex) {**

System.out.println("Error writing to server: " + ex.getMessage()); **}}}**

**Program for User Thread class:-**

import java.io.\*;

import java.net.\*;

import java.util.\*;

*// This thread handles connection for each connected client, so the server can handle multiple clients at //the same time.*

**public class UserThread extends Thread {**

private Socket socket;

private ChatServer server;

private PrintWriter writer;

**public UserThread(Socket socket, ChatServer server) {**

this.socket = socket;

this.server = server; **}**

**public void run() {**

**try {**

**InputStream input = socket.getInputStream();**

**BufferedReader reader = new BufferedReader(new InputStreamReader(input));**

**OutputStream output = socket.getOutputStream();**

writer = new PrintWriter(output, true);

printUsers();

String userName = reader.readLine();server.addUserName(userName);

String serverMessage = "New user connected: " + userName;

server.broadcast(serverMessage, this); String clientMessage;

**do {**

clientMessage = reader.readLine();

serverMessage = "[" + userName + "]: " + clientMessage;

server.broadcast(serverMessage, this);

**} while (!clientMessage.equals("bye"));**

server.removeUser(userName, this);

socket.close();

serverMessage = userName + " has quitted."; server.broadcast(serverMessage, this);

**} catch (IOException ex) {**

System.out.println("Error in UserThread: " + ex.getMessage());

ex.printStackTrace(); **}}**

*//Sends a list of online users to the newly connected user.*

**void printUsers() {**

**if (server.hasUsers()) {**

writer.println("Connected users: " + server.getUserNames());

**} else {**

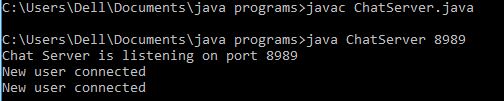
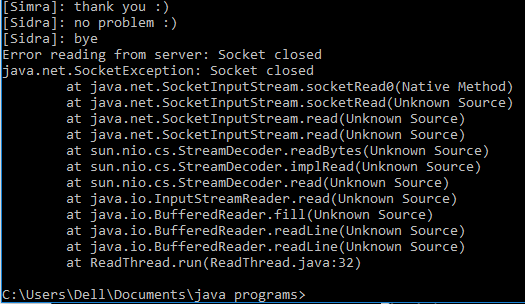
writer.println("No other users connected");  **}}**

*// Sends a message to the client.*

**void sendMessage(String message) {**

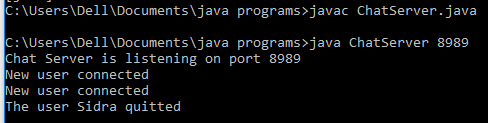
writer.println(message); **}}**

**OUTPUT OF SERVER:- (Server must be run first).**



**When the user(sidra)enters bye then**

**it will disconnect from the server.**

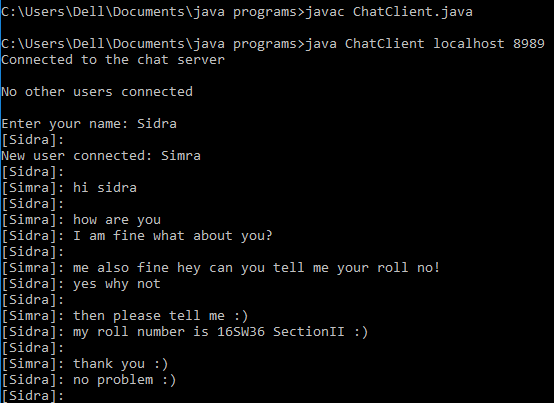
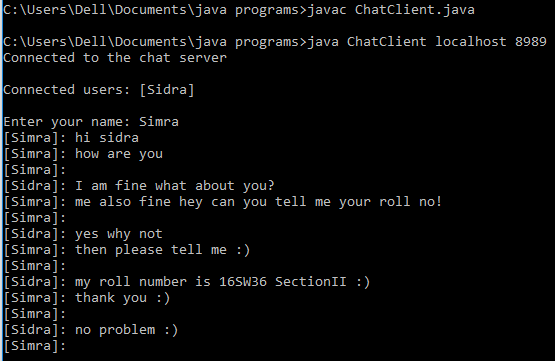
****

**And when user sidra disconnected**

**from the server then the output put**

**will be**

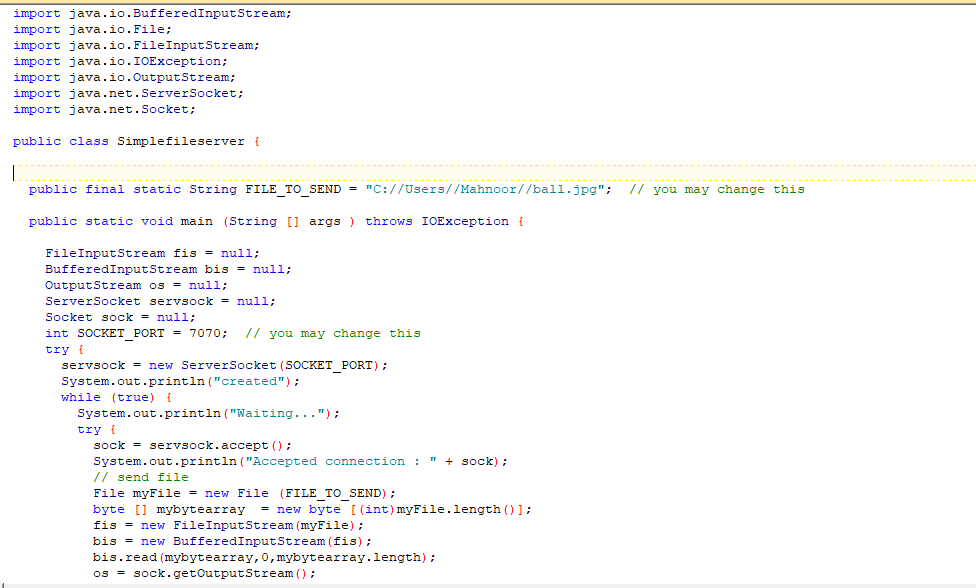
**OUTPUT OF ChatClient:-**

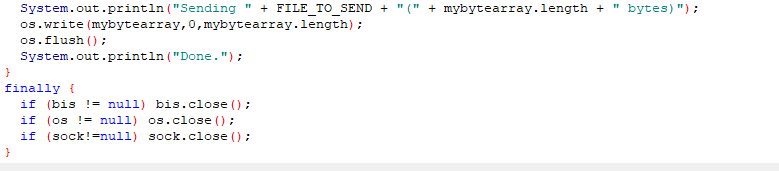


These are 2 users chatting with each other on different command prompt.As we can see in the 1st image Sidra is the first user and from the other image we can see the a new user is conneted with sidra that’s name is Simra and she starts chatting with her.

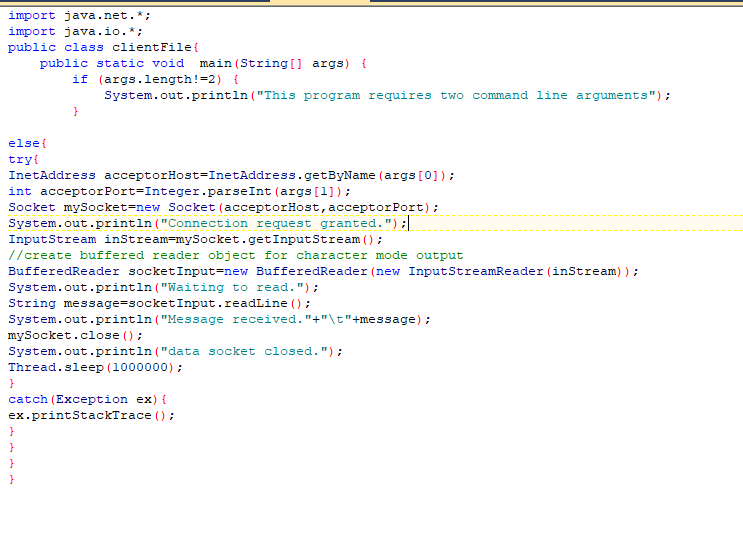
**Task # 05:-Modify the sample code to send complete files between the client to the server.**

**Server class:**

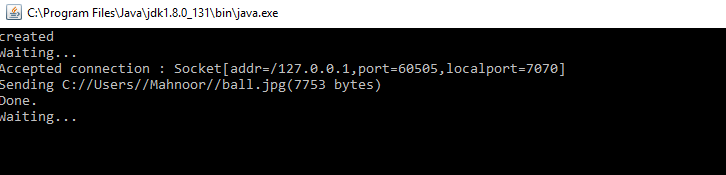
****

****

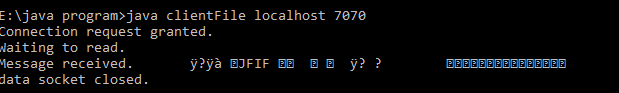
**Client class:**

****

**Server side result:**

****

**Client side result:**

****

**Task # 06:-Explore the non-blocking java socket API in the nio package and implement a sample program.**

Socket programs in [Java](http://www.developer.com/java) can be made to work in both blocking and non-blocking mode. In blocking socket mode, a system call event halts the execution until an appropriate reply has been received. In non-blocking sockets, it continues to execute even if the system call has been invoked and deals with its reply appropriately later.

## Non-blocking Sockets in Java

Java has TCP and UDP sockets. The methods such as connect(), accept(), read(), and write() defined in the ServerSocket and Socket class are used for blocking socket programming. For example, when a client invokes the read() method to read data from the server, the thread gets blocked until the data is available. This situation is undesirable under some circumstances. Instead, what we can do is use the waiting period to do some other task. The client socket then can notify when the data is available. Another problem is that, in a multi-socket connection, each client is a separate thread. Therefore, there is an overhead of maintaining a pool of client threads.

Blocking sockets are simple due to their sequential execution. Non-blocking sockets, on the other hand, are non-sequential. They require a different perspective to implement them in programming. In a way, non-blocking socket programs are a little complex and a bit more advanced technique of socket communication.

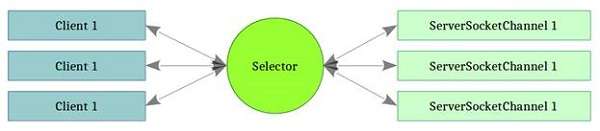
## Non-blocking Socket APIs

The classes that support non-blocking socket communication in Java are as follows.

|  |  |
| --- | --- |
| Non-blocking socket classes | Description (\*) |
| ServerSocketChannel | A selectable channel for stream-oriented listening sockets. The channel is created by invoking the open method of this class. This class uses the ServerSocket class behind the scenes. An instance of this class is used to accept a new connection request in a server like the ServerSocket class instance. |
| SocketChannel | A selectable channel for stream-oriented listening sockets. The channel is created by invoking the open method of this class. This class uses the Socket class behind the scenes. An instance of this class is used to establish connection between the client and the server like the Socket class instance. |
| Selector | A multiplexor for the SelectableChannel object. |
| SelectionKey | Used for representing the registration of SelectableChannel with a Selector. |

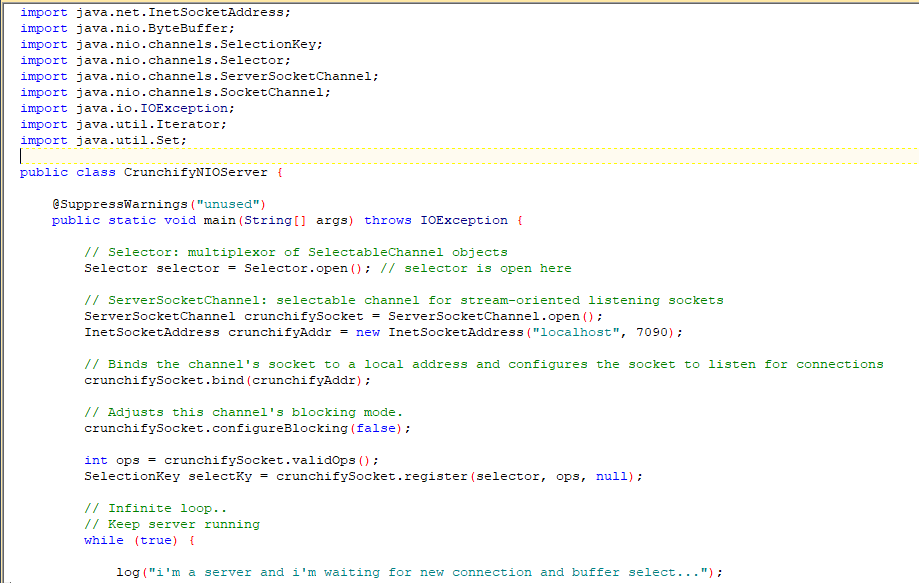
## How It Works

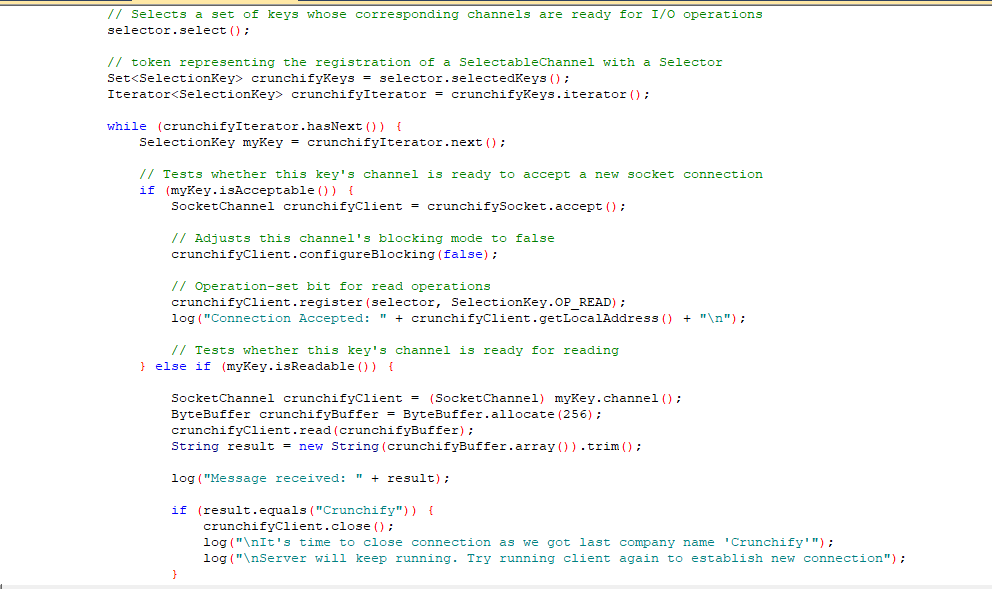
The Selector class object acts as an interface between remote clients and the server in the following manner.

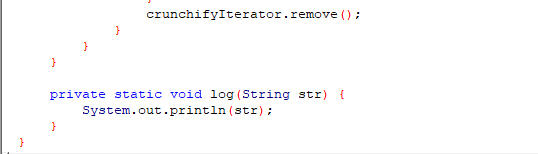


Sample Program:

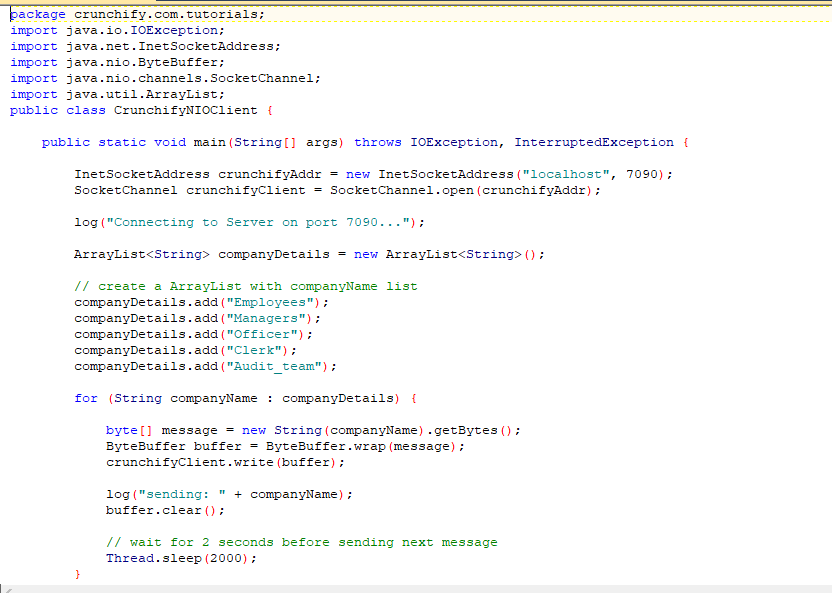
Server class:

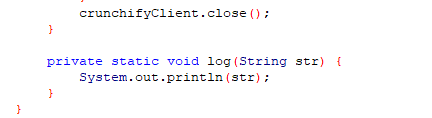




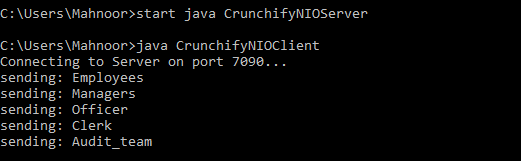


Client class:





Client side result:



Server side Result:

