

Department of Software Engineering
Mehran University of Engineering and Technology, Jamshoro

Course: SW422-Distributed Computing

Instructor	RabeeaJaffari	Practical/Lab No.	02
Date		CLOs	CLO-3: P5 & CLO-3: P3
Signature		Assessment Score	

Topic	To work with advanced client-server distributed applications
Objectives	- Learn to program advanced operations such as sending and receiving messages, filing etc. using stream sockets

Lab Discussion: Theoretical concepts and Procedural steps

Lab Tasks

Submission Date:

1. Compile and run the above code. Start the acceptor first and then the requestor with appropriate command line arguments. Describe and explain the output.

```

× ConnectionAcceptor.java × Server.java × Client.java ×
1 import java.net.*;
2 import java.io.*;
3 public class ConnectionAcceptor {
4     //Two command line arguments are needed
5     // port number of the server socket and second is the message to send
6     //16SW04 Jawaria Dhakhan
7     public static void main(String[] args){
8         if(args.length!=2){
9             System.out.println("This program requires two command line arguments");
10        }
11        else{
12
13            try{
14                int portNo=Integer.parseInt(args[0]);
15                String message=args[1];
16                ServerSocket connectionSocket=new ServerSocket(portNo);
17                System.out.println("now ready to accept a connection");
18                Socket dataSocket=connectionSocket.accept();
19                System.out.println("Connection Accepted");
20                OutputStream outputStream=dataSocket.getOutputStream();
21                //create a print writer for character mode output
22                PrintWriter socketOutput=new PrintWriter(new OutputStreamWriter(outputStream));
23                Thread.sleep(5000); //delay of 5 seconds
24                //write a message into the data stream
25                socketOutput.println(message);
26
27                //the ensuing flush method
28                //ensures that data is written into the data socket before the socket is closed.
29                socketOutput.flush();
30                System.out.println("message sent");
31                dataSocket.close();
32                System.out.println("data socket closed.");
33                connectionSocket.close();
34                System.out.println("connection socket closed.");
35                Thread.sleep(10000);
36            }
37            catch(Exception ex){ ex.printStackTrace();
38            } } }

```

```

1 import java.net.*;
2 import java.io.*;
3 //this application requests a connection and sends a message
4 // using the stream mode socket.
5 public class ConnectionRequestor {
6 public static void main(String[] args){
7 if(args.length!=2){
8 System.out.println("This program requires two command line arguments");
9 // the arguments are
10 //host name of connection acceptor and port number of connection acceptor
11 //16SW04 Jawaria Dhakhan
12 }
13 else{
14 try{
15 InetAddress acceptorHost=InetAddress.getByName(args[0]);
16 int acceptorPort=Integer.parseInt(args[1]);
17 Socket mySocket=new Socket(acceptorHost,acceptorPort);
18 System.out.println("Connection request granted.");
19 InputStream inStream=mySocket.getInputStream();
20 //create buffered reader object for character mode output
21 BufferedReader socketInput=new BufferedReader(new InputStreamReader(inStream));
22 System.out.println("Waiting to read.");
23 String message=socketInput.readLine();
24 System.out.println("Message received."+ "\t"+message);
25 mySocket.close();
26 System.out.println("data socket closed.");
27 Thread.sleep(10000);
28 }
29 catch(Exception ex){
30 ex.printStackTrace();
31 }}}}
--

```

E:\8th Semester\DC Labs\16SW04\Lab2>start java ConnectionAcceptor 9090 16SW04

E:\8th Semester\DC Labs\16SW04\Lab2>java ConnectionRequestor 192.168.8.20 9090

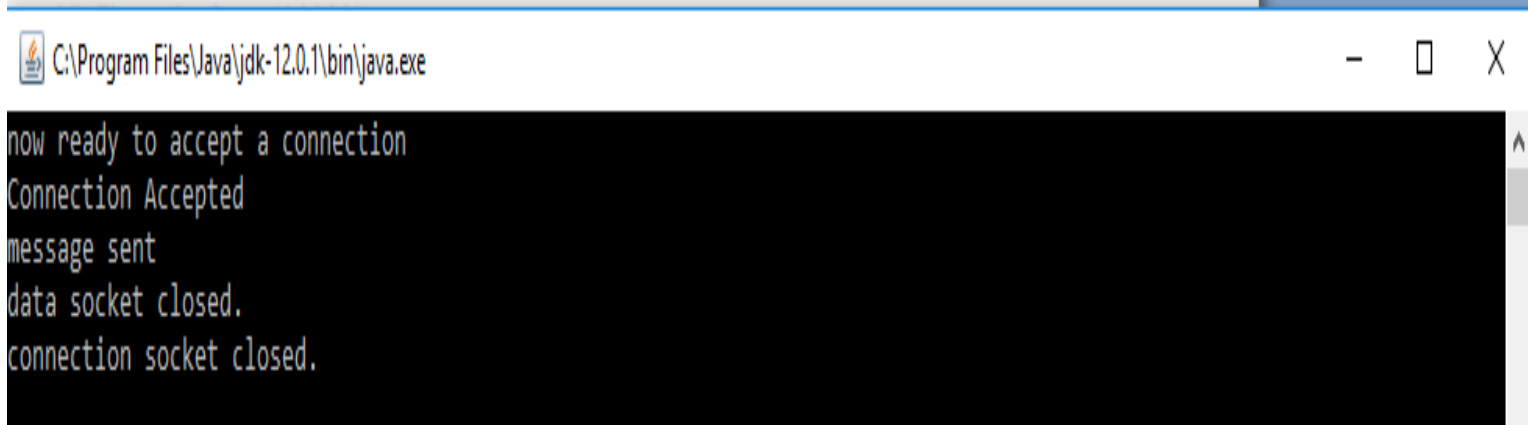
Connection request granted.

Waiting to read.

Message received. 16SW04

data socket closed.

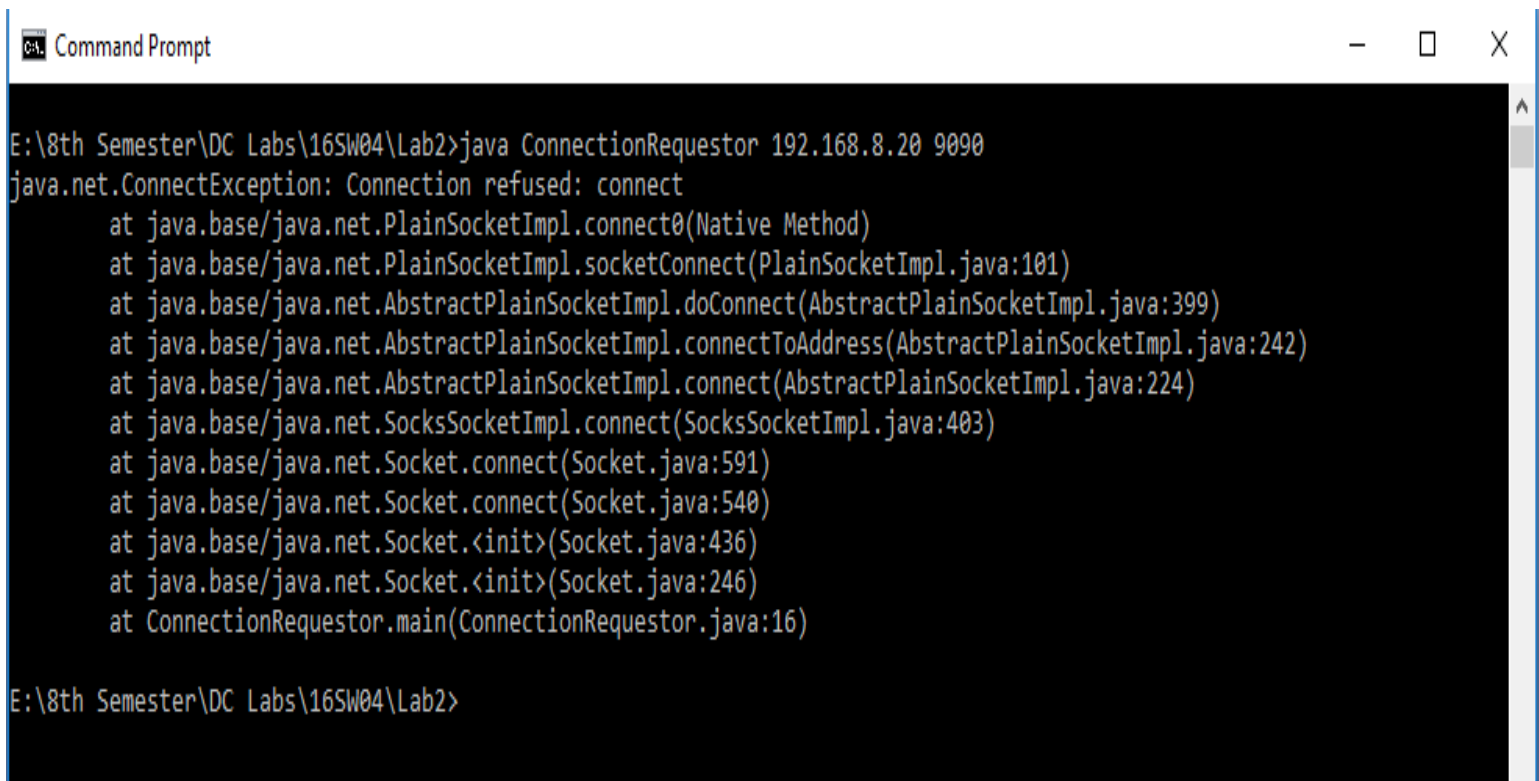
E:\8th Semester\DC Labs\16SW04\Lab2>



```
now ready to accept a connection
Connection Accepted
message sent
data socket closed.
connection socket closed.
```

ConnectionAcceptor accepts a connection and sends message to the ConnectionRequestor. ConnectionRequestor receives message then connection is closed.

2. Now run the code again, but reverse the order of program's execution. Start the requestor first and then the acceptor. Describe and explain the outcome.



```
E:\8th Semester\DC Labs\16SW04\Lab2>java ConnectionRequestor 192.168.8.20 9090
java.net.ConnectException: Connection refused: connect
    at java.base/java.net.PlainSocketImpl.connect0(Native Method)
    at java.base/java.net.PlainSocketImpl.socketConnect(PlainSocketImpl.java:101)
    at java.base/java.net.AbstractPlainSocketImpl.doConnect(AbstractPlainSocketImpl.java:399)
    at java.base/java.net.AbstractPlainSocketImpl.connectToAddress(AbstractPlainSocketImpl.java:242)
    at java.base/java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:224)
    at java.base/java.net.SocksSocketImpl.connect(SocksSocketImpl.java:403)
    at java.base/java.net.Socket.connect(Socket.java:591)
    at java.base/java.net.Socket.connect(Socket.java:540)
    at java.base/java.net.Socket.<init>(Socket.java:436)
    at java.base/java.net.Socket.<init>(Socket.java:246)
    at ConnectionRequestor.main(ConnectionRequestor.java:16)

E:\8th Semester\DC Labs\16SW04\Lab2>
```

If we start the Requestor first, the program ends up with an exception of `ConnectException`. Since, Connection is not established.

3. 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.

```
ConnectionAcceptor.java*
1 import java.net.*;
2 import java.io.*;
3 public class ConnectionAcceptor {
4     //Two command line arguments are needed
5     // port number of the server socket and second is the message to send
6     //16SW04 Jawaria Dhakhan
7     public static void main(String[] args){
8         if(args.length!=2){
9             System.out.println("This program requires two command line arguments");
10        }
11        else{
12
13            try{
14                int portNo=Integer.parseInt(args[0]);
15                String message=args[1];
16                ServerSocket connectionSocket=new ServerSocket(portNo);
17                System.out.println("now ready to accept a connection");
18                Socket dataSocket=connectionSocket.accept();
19                System.out.println("Connection Accepted");
20                OutputStream outputStream=dataSocket.getOutputStream();
21                //create a print writer for character mode output
22                PrintWriter socketOutput=new PrintWriter(new OutputStreamWriter(outputStream));
23                Thread.sleep(5000);//delay of 5 seconds
24                //write a message into the data stream
25                socketOutput.println(message);
26
27                //the ensuing flush method
28                //ensures that data is written into the data socket before the socket is closed.
29                socketOutput.flush();
30                System.out.println("message sent");
31                dataSocket.close();
32                System.out.println("data socket closed.");
33                connectionSocket.close();
34                System.out.println("connection socket closed.");
35                Thread.sleep(10000);
36            }
37            catch(Exception ex){ ex.printStackTrace();
38            } } }
```

Command Prompt - java ConnectionRequester 192.168.8.20 9090

E:\8th Semester\DC Labs\16SW04\Lab2>start java ConnectionAcceptor 9090 16SW04

E:\8th Semester\DC Labs\16SW04\Lab2>java ConnectionRequester 192.168.8.20 9090

Connection request granted.

Waiting to read.

```

E:\8th Semester\DC Labs\16SW04\Lab2>start java ConnectionAcceptor 9090 16SW04
E:\8th Semester\DC Labs\16SW04\Lab2>java ConnectionRequestor 192.168.8.20 9090
Connection request granted.
Waiting to read.
Message received.      16SW04
data socket closed.

E:\8th Semester\DC Labs\16SW04\Lab2>

```

Bonus Tasks

Lab 02 (+1)

1. Modify the sample code to include two way communication between the client and the server.

```

Server.java
Client.java

1 import java.io.*;
2 import java.net.*;
3 public class Server
4 {
5     public static void main(String[] args) throws Exception
6     {
7         ServerSocket sersock = new ServerSocket(3000);
8         System.out.println("Server ready for chatting");
9         Socket sock = sersock.accept();
10        //16SW04 Jawaria Dhakhan
11        // reading from keyboard (keyRead object)
12        BufferedReader keyRead = new BufferedReader(new InputStreamReader(System.in));
13        // sending to client (pwrite object)
14        OutputStream ostream = sock.getOutputStream();
15        PrintWriter pwrite = new PrintWriter(ostream, true);
16
17        // receiving from server ( receiveRead object)
18        InputStream istream = sock.getInputStream();
19        BufferedReader receiveRead = new BufferedReader(new InputStreamReader(istream));
20
21        String receiveMessage, sendMessage;
22        while(true)
23        {
24            if((receiveMessage = receiveRead.readLine()) != null)
25            {
26                System.out.print("Client: ");
27                System.out.println(receiveMessage);
28            }
29            sendMessage = keyRead.readLine();
30            pwrite.println(sendMessage);
31            pwrite.flush();
32        }
33    }
34 }

```

```
1 import java.io.*;
2 import java.net.*;
3 public class Client
4 {
5     public static void main(String[] args) throws Exception
6     {
7         Socket sock = new Socket("127.0.0.1", 3000);
8         // reading from keyboard (keyRead object)
9         BufferedReader keyRead = new BufferedReader(new InputStreamReader(System.in));
10        // sending to client (pwrite object)
11        //16SW04 Jawaria Dhakhan
12        OutputStream ostream = sock.getOutputStream();
13        PrintWriter pwrite = new PrintWriter(ostream, true);
14
15        // receiving from server ( receiveRead object)
16        InputStream istream = sock.getInputStream();
17        BufferedReader receiveRead = new BufferedReader(new InputStreamReader(istream));
18
19        System.out.println("Start the chitchat with Server, type and press Enter key");
20
21        String receiveMessage, sendMessage;
22        while(true)
23        {
24            sendMessage = keyRead.readLine(); // keyboard reading
25            pwrite.println(sendMessage); // sending to server
26            pwrite.flush(); // flush the data
27            if((receiveMessage = receiveRead.readLine()) != null) //receive from server
28            {
29                System.out.print("Server: ");
30                System.out.println(receiveMessage); // displaying at DOS prompt
31            }
32        }
33    }
34 }
```

```
Command Prompt - java Client

E:\8th Semester\DC Labs\16SW04\Lab2>javac *.java

E:\8th Semester\DC Labs\16SW04\Lab2>start java Server

E:\8th Semester\DC Labs\16SW04\Lab2>java Client
Start the chitchat with Server, type and press Enter key
Hello Server, I am Jawaria Dhakhan
Server: How are you, 16SW04
I am good, Alhamdiallah:)
Server: You have completed your DC Lab 02, Hurray!
Thanks:D
```

```
C:\Program Files\Java\jdk-12.0.1\bin\java.exe

Server ready for chatting
Client: Hello Server, I am Jawaria Dhakhan
How are you, 16SW04
Client: I am good, Alhamdiallah:)
You have completed your DC Lab 02, Hurray!
Client: Thanks:D
```


2. Modify the sample code to send complete files between the client to the server.

- **Sending text files to client.**

```
ClientFile.java  ServerFile.java

1 import java.io.BufferedInputStream;
2 import java.io.File;
3 import java.io.FileInputStream;
4 import java.io.IOException;
5 import java.io.OutputStream;
6 import java.net.ServerSocket;
7 import java.net.Socket;
8
9 public class ServerFile {
10     //16SW04 Jawaria Dhakhan
11     public final static int SOCKET_PORT = Integer.parseInt("8080");
12     public final static String FILE_TO_SEND = "C:/Users/KING SAMOON/Desktop/Jawaria.txt";
13
14     public static void main (String [] args ) throws IOException {
15
16         FileInputStream fis = null;
17         BufferedInputStream bis = null;
18         OutputStream os = null;
19         ServerSocket servsock = null;
20         Socket sock = null;
21         try {
22             servsock = new ServerSocket(SOCKET_PORT);
23             while (true) {
24                 System.out.println("Waiting...");
25                 try {
26                     sock = servsock.accept();
27                     System.out.println("Accepted connection : " + sock);
28                     // send file
29                     File myFile = new File (FILE_TO_SEND);
30                     byte [] mybytearray = new byte [(int)myFile.length()];
31                     fis = new FileInputStream(myFile);
32                     bis = new BufferedInputStream(fis);
33                     bis.read(mybytearray,0,mybytearray.length);
34                     os = sock.getOutputStream();
35                     System.out.println("Sending " + FILE_TO_SEND + "(" + mybytearray.length + " bytes)");
36                     os.write(mybytearray,0,mybytearray.length);
37                     os.flush();
38                     System.out.println("Done.");
39                 }
40                 finally {
41                     if (bis != null) bis.close();
42                     if (os != null) os.close();
43                     if (sock!=null) sock.close();}}}
44         finally {
45             if (servsock != null) servsock.close();
46     }}
```



```
Command Prompt

E:\8th Semester\DC Labs\16SW04\Lab2>javac ServerFile.java

E:\8th Semester\DC Labs\16SW04\Lab2>start java ServerFile

E:\8th Semester\DC Labs\16SW04\Lab2>.
```

```
C:\Program Files\Java\jdk-12.0.1\bin\java.exe

Waiting...
Accepted connection : Socket[addr=/192.168.8.43,port=49760,localport=8080]
Sending C:/Users/KING SAMOON/Desktop/Jawaria.txt(7 bytes)
Done.
Waiting...
```

```
ClientFile.java  ServerFile.java

1 import java.io.BufferedOutputStream;
2 import java.io.FileOutputStream;
3 import java.io.IOException;
4 import java.io.InputStream;
5 import java.net.*;
6
7 public class ClientFile {
8
9     public final static String
10         FILE_TO_RECEIVED = "C:/Users/Haier/Desktop/16SW04/JawariaDhakhani.txt";
11
12     public final static int FILE_SIZE = 6022386;
13     //16SW04 Jawaria Dhakhani
14     public static void main (String [] args ) throws IOException {
15         int bytesRead;
16         int current = 0;
17         FileOutputStream fos = null;
18         BufferedOutputStream bos = null;
19         Socket sock = null;
20         try {
21
22             InetAddress SERVERHOST = InetAddress.getByName("192.168.8.20");
23             int SOCKET_PORT = 8080;
24
25             sock = new Socket(SERVERHOST, SOCKET_PORT);
26             System.out.println("Connected...");
27
28             // receive file
29             byte [] mybytearray = new byte [FILE_SIZE];
30             InputStream is = sock.getInputStream();
31             fos = new FileOutputStream(FILE_TO_RECEIVED);
32             bos = new BufferedOutputStream(fos);
33             bytesRead = is.read(mybytearray,0,mybytearray.length);
34             current = bytesRead;
35
```

```
ClientFile.java  ServerFile.java
35
36
37     do {
38         bytesRead =
39             is.read(mybytearray, current, (mybytearray.length-current));
40         if(bytesRead >= 0) current += bytesRead;
41     } while(bytesRead > -1);
42
43     bos.write(mybytearray, 0 , current);
44     bos.flush();
45     System.out.println("File " + FILE_TO_RECEIVED
46         + " downloaded (" + current + " bytes read)");
47 }
48 finally {
49     if (fos != null) fos.close();
50     if (bos != null) bos.close();
51     if (sock != null) sock.close();
52 }
53 }
54
55 }
```

```
Command Prompt
C:\Users\Haier\Desktop\16SW04>javac FileClient.java
C:\Users\Haier\Desktop\16SW04>java FileClient
Connected...
File C:/Users/Haier/Desktop/16SW04/JawariaDhakhan.txt downloaded (7 bytes read)
C:\Users\Haier\Desktop\16SW04>
```

- Sending image files to client.

```

ClientFile.java  ServerFile.java
1 import java.io.BufferedInputStream;
2 import java.io.File;
3 import java.io.FileInputStream;
4 import java.io.IOException;
5 import java.io.OutputStream;
6 import java.net.ServerSocket;
7 import java.net.Socket;
8
9 public class ServerFile {
10     //16SW04 Jawaria Dhakhan
11     public final static int SOCKET_PORT = Integer.parseInt("8080");
12     public final static String FILE_TO_SEND = "C:/Users/KING SAMOON/Desktop/Jawaria.jpg";
13
14     public static void main (String [] args ) throws IOException {
15
16         FileInputStream fis = null;
17         BufferedInputStream bis = null;
18         OutputStream os = null;
19         ServerSocket servsock = null;
20         Socket sock = null;
21         try {
22             servsock = new ServerSocket(SOCKET_PORT);
23             while (true) {
24                 System.out.println("Waiting...");
25                 try {
26                     sock = servsock.accept();
27                     System.out.println("Accepted connection : " + sock);
28                     // send file
29                     File myFile = new File (FILE_TO_SEND);
30                     byte [] mybytearray = new byte [(int)myFile.length()];
31                     fis = new FileInputStream(myFile);
32                     bis = new BufferedInputStream(fis);
33                     bis.read(mybytearray,0,mybytearray.length);
34                     os = sock.getOutputStream();
35                     System.out.println("Sending " + FILE_TO_SEND + "(" + mybytearray.length + " bytes)");
36                     os.write(mybytearray,0,mybytearray.length);
37                     os.flush();
38                     System.out.println("Done.");
39                 }
40                 finally {
41                     if (bis != null) bis.close();
42                     if (os != null) os.close();
43                     if (sock!=null) sock.close();}}}
44             finally {
45                 if (servsock != null) servsock.close();
46             }
47         }
48     }
49 }

```

```

Command Prompt
E:\8th Semester\DC Labs\16SW04\Lab2>javac ServerFile.java
E:\8th Semester\DC Labs\16SW04\Lab2>start java ServerFile
E:\8th Semester\DC Labs\16SW04\Lab2>

```

C:\Program Files\Java\jdk-12.0.1\bin\java.exe

Waiting...

Accepted connection : Socket[addr=/192.168.8.43,port=49773,localport=8080]

Sending C:/Users/KING SAMOON/Desktop/Jawaria.jpg(63506 bytes)

Done.

Waiting...

```
ClientFile.java  ServerFile.java
1 import java.io.BufferedOutputStream;
2 import java.io.FileOutputStream;
3 import java.io.IOException;
4 import java.io.InputStream;
5 import java.net.*;
6
7 public class ClientFile {
8
9     public final static String
10         FILE_TO_RECEIVED = "C:/Users/Haier/Desktop/16SW04/JawariaDhakhan.jpg";
11
12     public final static int FILE_SIZE = 6022386;
13     //16SW04 Jawaria Dhakhan
14     public static void main (String [] args ) throws IOException {
15         int bytesRead;
16         int current = 0;
17         FileOutputStream fos = null;
18         BufferedOutputStream bos = null;
19         Socket sock = null;
20         try {
21
22             InetAddress SERVERHOST = InetAddress.getByName("192.168.8.20");
23             int SOCKET_PORT = 8080;
24
25             sock = new Socket(SERVERHOST, SOCKET_PORT);
26             System.out.println("Connected...");
27
28             // receive file
29             byte [] mybytearray = new byte [FILE_SIZE];
30             InputStream is = sock.getInputStream();
31             fos = new FileOutputStream(FILE_TO_RECEIVED);
32             bos = new BufferedOutputStream(fos);
33             bytesRead = is.read(mybytearray,0,mybytearray.length);
34             current = bytesRead;
35
36         }
37     }
38 }
```

```
ClientFile.java  ServerFile.java
36
37     do {
38         bytesRead =
39             is.read(mybytearray, current, (mybytearray.length-current));
40         if(bytesRead >= 0) current += bytesRead;
41     } while(bytesRead > -1);
42
43     bos.write(mybytearray, 0 , current);
44     bos.flush();
45     System.out.println("File " + FILE_TO_RECEIVED
46         + " downloaded (" + current + " bytes read)");
47 }
48 finally {
49     if (fos != null) fos.close();
50     if (bos != null) bos.close();
51     if (sock != null) sock.close();
52 }
53 }
54
55 }
```

```
Command Prompt
C:\Users\Haier\Desktop\16SW04>javac FileClient.java
C:\Users\Haier\Desktop\16SW04>java FileClient
Connected...
File C:/Users/Haier/Desktop/16SW04/JawariaDhakhan.jpg downloaded (63506 bytes read)
C:\Users\Haier\Desktop\16SW04>_
```

3. Explore the non-blocking java socket API in the **nio** package and implement a sample program.

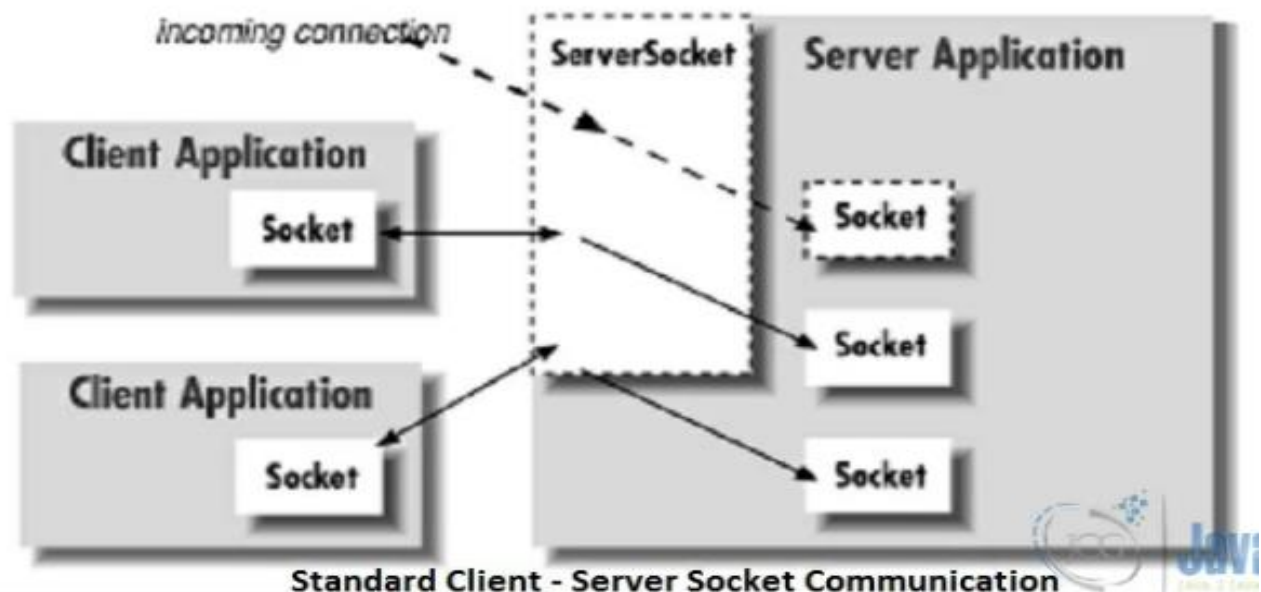
Non-blocking Sockets in Java:

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.

- **Standard Java sockets**

Socket programming involves two systems communicating with one another. In implementations prior to NIO, Java TCP client socket code is handled by the `java.net.Socket` class. A socket is one end-point of a two-way communication link between two programs running on the network. Socket classes are used to represent the connection between a client program and a server program. The `java.net` package provides two classes, `Socket` and `ServerSocket`, that implement the client side of the connection and the server side of the connection, respectively. The below image illustrates the nature of this communication:



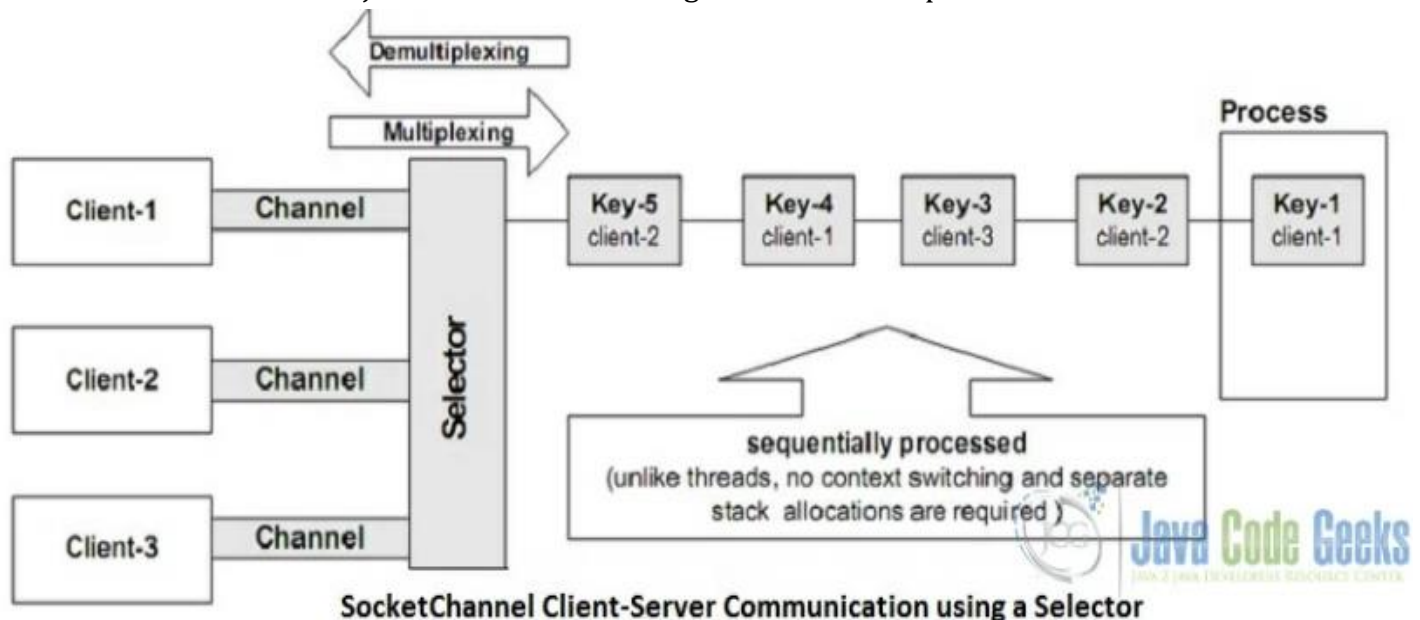
A socket is basically a blocking input/output device. It makes the thread that is using it to block on reads and potentially also block on writes if the underlying buffer is full. Therefore, different threads are required if the server has many open sockets. From a simplistic perspective, the process of a blocking socket communication is as follows:

- Create a ServerSocket, specifying a port to listen on.
- Invoke the ServerSocket's accept() method to listen on the configured port for a client connection.
- When a client connects to the server, the accept() method returns a Socket through which the server can communicate with the client: an InputStream is obtained to read from the client and an OutputStream to write to the client.

- **Non-blocking SocketChannel with java.nio**

With the standard java sockets, if the server needed to be scalable, the socket had to be passed to another thread for processing so that the server could continue listening for additional connections, meaning call the ServerSocket's accept() method again to listen for another connection.

A SocketChannel on the other hand is a non-blocking way to read from sockets, so that you can have one thread communicate with multiple open connections at once. With socket channel we describe the communication channel between client and server. It is identified by the server IP address and the port number. Data passes through the socket channel by buffer items. A selector monitors the recorded socket channels and serializes the requests, which the server has to satisfy. The Keys describe the objects used by the selector to sort the requests. Each key represents a single client sub-request and contains information to identify the client and the type of the request. With non-blocking I/O, someone can program networked applications to handle multiple simultaneous connections without having to manage multiple thread collection, while also taking advantage of the new server scalability that is built in to java.nio. The below image illustrates this procedure:



- Sample Program

```
SocketServer.java SocketClient.java
1 import java.io.IOException;
2 import java.net.InetSocketAddress;
3 import java.net.Socket;
4 import java.net.SocketAddress;
5 import java.nio.ByteBuffer;
6 import java.nio.channels.SelectionKey;
7 import java.nio.channels.Selector;
8 import java.nio.channels.ServerSocketChannel;
9 import java.nio.channels.SocketChannel;
10 import java.util.*;
11
12 public class SocketServer {
13     private Selector selector;
14     private Map<SocketChannel, List> dataMapper;
15     private InetSocketAddress listenAddress;
16     // 16SW04 JAWARIA DHAKHAN
17     public static void main(String[] args) throws Exception {
18         Runnable server = new Runnable() {
19             @Override
20             public void run() {
21                 try {
22                     new SocketServer("localhost", 8090).startServer();
23                 } catch (IOException e) {
24                     e.printStackTrace();
25                 }
26             }
27         };
28         Runnable client = new Runnable() {
29             @Override
30             public void run() {
31                 try {
32                     new SocketClient().startClient();
33                 } catch (IOException e) {
34                     e.printStackTrace();
35                 } catch (InterruptedException e) {
36                     e.printStackTrace();
37                 }
38             }
39         };
40     }
41 }
```

```
39         }
40
41     }
42 };
43 new Thread(server).start();
44 new Thread(client, "client-A").start();
45 new Thread(client, "client-B").start();
46 }
47
48 public SocketServer(String address, int port) throws IOException {
49     listenAddress = new InetSocketAddress(address, port);
50     dataMapper = new HashMap<SocketChannel, List>();
51 }
52
53 // create server channel
54 private void startServer() throws IOException {
55     this.selector = Selector.open();
56     ServerSocketChannel serverChannel = ServerSocketChannel.open();
57     serverChannel.configureBlocking(false);
58
59     // retrieve server socket and bind to port
60     serverChannel.socket().bind(listenAddress);
61     serverChannel.register(this.selector, SelectionKey.OP_ACCEPT);
62
63     System.out.println("Server started...");
64
65     while (true) {
66         // wait for events
67         this.selector.select();
68
69         //work on selected keys
70         Iterator keys = this.selector.selectedKeys().iterator();
71         while (keys.hasNext()) {
72             SelectionKey key = (SelectionKey) keys.next();
73
74             // this is necessary to prevent the same key from coming up
75             // again the next time around.
76             keys.remove();
```

```
77
78         if (!key.isValid()) {
79             continue;
80         }
81
82         if (key.isAcceptable()) {
83             this.accept(key);
84         }
85         else if (key.isReadable()) {
86             this.read(key);
87         }
88     }
89 }
90
91
92 //accept a connection made to this channel's socket
93 private void accept(SelectionKey key) throws IOException {
94     ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();
95     SocketChannel channel = serverChannel.accept();
96     channel.configureBlocking(false);
97     Socket socket = channel.socket();
98     SocketAddress remoteAddr = socket.getRemoteSocketAddress();
99     System.out.println("Connected to: " + remoteAddr);
100
101     // register channel with selector for further IO
102     dataMapper.put(channel, new ArrayList());
103     channel.register(this.selector, SelectionKey.OP_READ);
104 }
105
106 //read from the socket channel
107 private void read(SelectionKey key) throws IOException {
108     SocketChannel channel = (SocketChannel) key.channel();
109     ByteBuffer buffer = ByteBuffer.allocate(1024);
110     int numRead = -1;
111     numRead = channel.read(buffer);
112
113     if (numRead == -1) {
114         this.dataMapper.remove(channel);
115
116         Socket socket = channel.socket();
117         SocketAddress remoteAddr = socket.getRemoteSocketAddress();
118         System.out.println("Connection closed by client: " + remoteAddr);
119         channel.close();
120         key.cancel();
121         return;
122     }
123
124     byte[] data = new byte[numRead];
125     System.arraycopy(buffer.array(), 0, data, 0, numRead);
126     System.out.println("Got: " + new String(data));
127 }
```

From the above code:

In the `main()` method on lines 43-45, one thread for creating the `ServerSocketChannel` is started and two client threads responsible for starting the clients which will create a `SocketChannel` for sending messages to the server.

- `new Thread(server).start();`
- `new Thread(client, "client-A").start();`
- `new Thread(client, "client-B").start();`

In the `startServer()` method on line 54, the server `SocketChannel` is created as `nonBlocking`, the server socket is retrieved and bound to the specified port:

- `ServerSocketChannel serverChannel = ServerSocketChannel.open();`
- `serverChannel.configureBlocking(false);`
 `// retrieve server socket and bind to port`
- `serverChannel.socket().bind(listenAddress);`

Finally, the `register` method associates the selector to the socket channel.

- `serverChannel.register(this.selector, SelectionKey.OP_ACCEPT);`

The second parameter represents the type of the registration. In this case, we use `OP_ACCEPT`, which means the selector merely reports that a client attempts a connection to the server. Other possible options are: `OP_CONNECT`, which will be used by the client; `OP_READ`; and `OP_WRITE`.

After that, the `select` method is used on line 67, which blocks the execution and waits for events recorded on the selector in an infinite loop.

- `this.selector.select();`

The selector waits for events and creates the keys. According to the key-types, an opportune operation is performed. There are four possible types for a key:

- **Acceptable:** the associated client requests a connection.
- **Connectable:** the server accepted the connection.
- **Readable:** the server can read.
- **Writable:** the server can write.

If an acceptable key is found, the `accept(SelectionKey key)` on line 93 is invoked in order to create a channel which accepts this connection, creates a standard java socket on line 97 and register the channel with the selector:

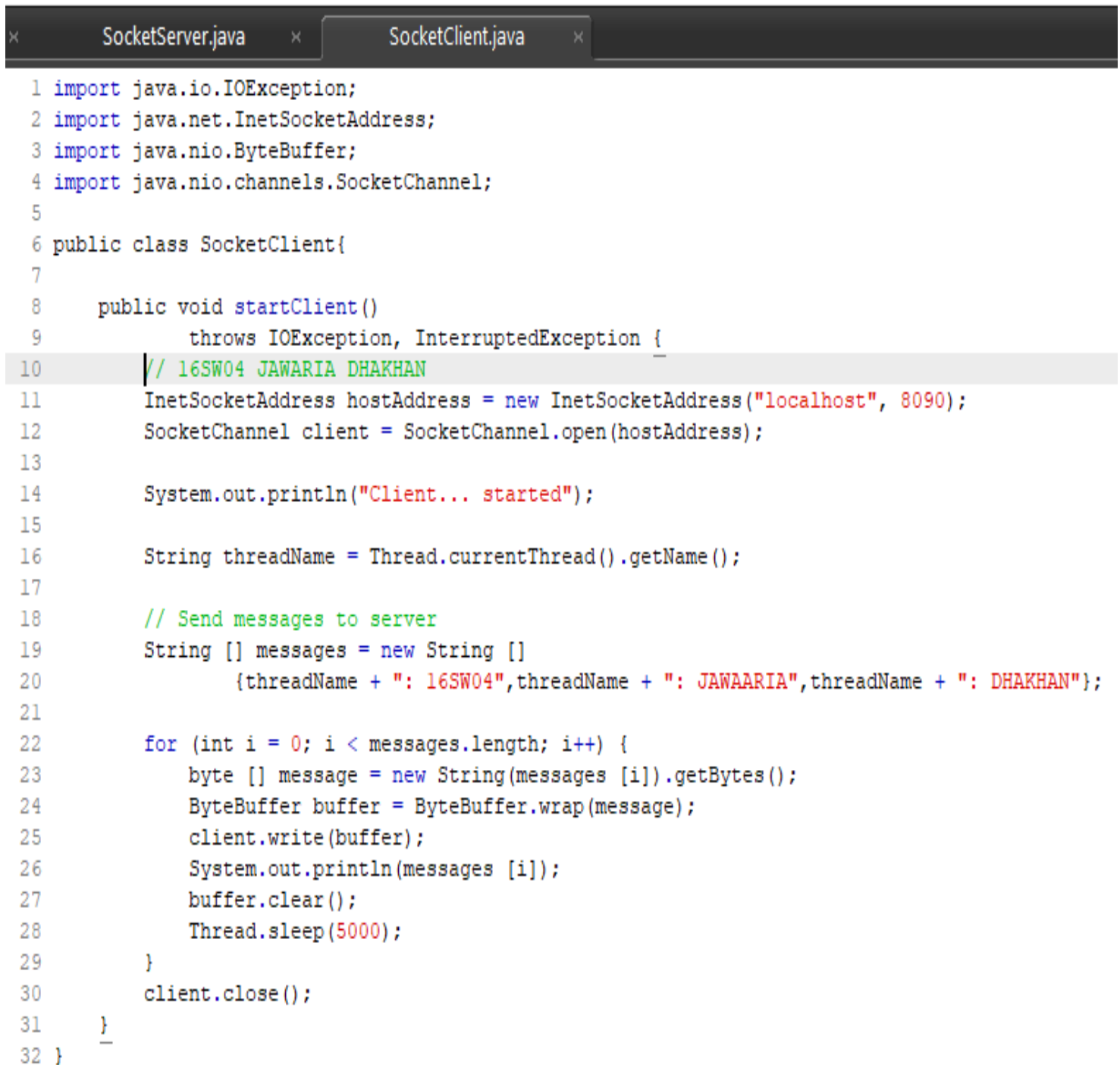
- `ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();`
- `SocketChannel channel = serverChannel.accept();`
- `channel.configureBlocking(false);`
- `Socket socket = channel.socket();`
- `SocketAddress remoteAddr = socket.getRemoteSocketAddress();`

After receiving a readable key from the client, the read(SelectionKey key) is called on line 107 which reads from the socket channel. A byte buffer is allocated for reading from the channel

- numRead = channel.read(buffer);

and the client's transmitted data are echoed on System.out:

- System.out.println("Got: " + new String(data));

A screenshot of a Java IDE with two tabs: 'SocketServer.java' and 'SocketClient.java'. The 'SocketClient.java' tab is active, showing a Java class with imports for java.io.IOException, java.net.InetSocketAddress, java.nio.ByteBuffer, and java.nio.channels.SocketChannel. The class SocketClient has a startClient method that throws IOException and InterruptedException. Inside startClient, it creates an InetSocketAddress for 'localhost' on port 8090, opens a SocketChannel, and prints 'Client... started'. It then creates an array of messages with thread names and IDs. A for loop iterates over these messages, wraps each in a ByteBuffer, writes it to the client, prints the message, clears the buffer, and sleeps for 5000ms. Finally, it closes the client.

```
1 import java.io.IOException;
2 import java.net.InetSocketAddress;
3 import java.nio.ByteBuffer;
4 import java.nio.channels.SocketChannel;
5
6 public class SocketClient{
7
8     public void startClient()
9         throws IOException, InterruptedException {
10         // 16SW04 JAWARIA DHAKHAN
11         InetSocketAddress hostAddress = new InetSocketAddress("localhost", 8090);
12         SocketChannel client = SocketChannel.open(hostAddress);
13
14         System.out.println("Client... started");
15
16         String threadName = Thread.currentThread().getName();
17
18         // Send messages to server
19         String [] messages = new String []
20             {threadName + ": 16SW04",threadName + ": JAWARIA",threadName + ": DHAKHAN"};
21
22         for (int i = 0; i < messages.length; i++) {
23             byte [] message = new String(messages [i]).getBytes();
24             ByteBuffer buffer = ByteBuffer.wrap(message);
25             client.write(buffer);
26             System.out.println(messages [i]);
27             buffer.clear();
28             Thread.sleep(5000);
29         }
30         client.close();
31     }
32 }
```

In the above client code,

each client thread creates a socket channel on the server's host address on line 12:

- `SocketChannel client = SocketChannel.open(hostAddress);`

On line 19, a String array is created to be transmitted to the server using the previously created socket. The data contain also each thread's name for distinguishing the sender:

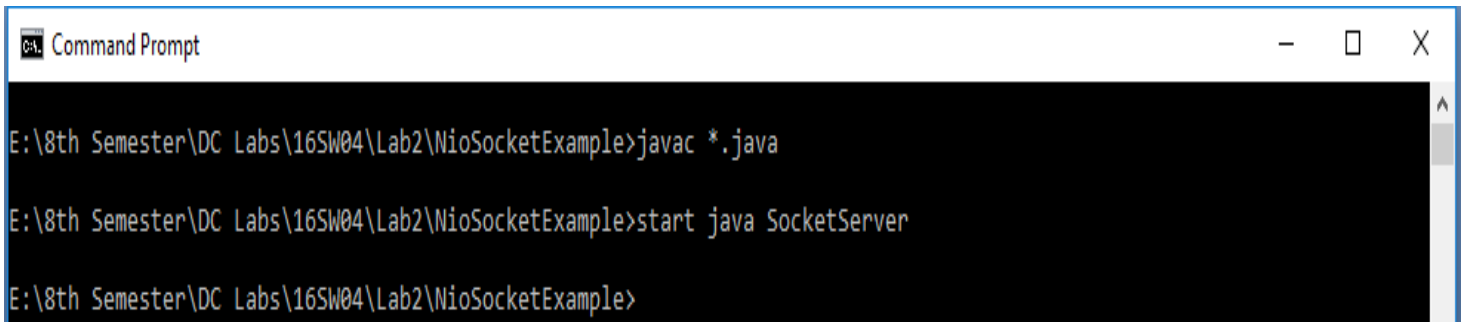
- `String threadName = Thread.currentThread().getName();`
`// Send messages to server`
- `String [] messages = new String []`
- `{threadName + ": 16SW04",threadName + ": JAWAARIA",threadName + ": DHAKHAN"};`

For each string message a buffer is created on line 24:

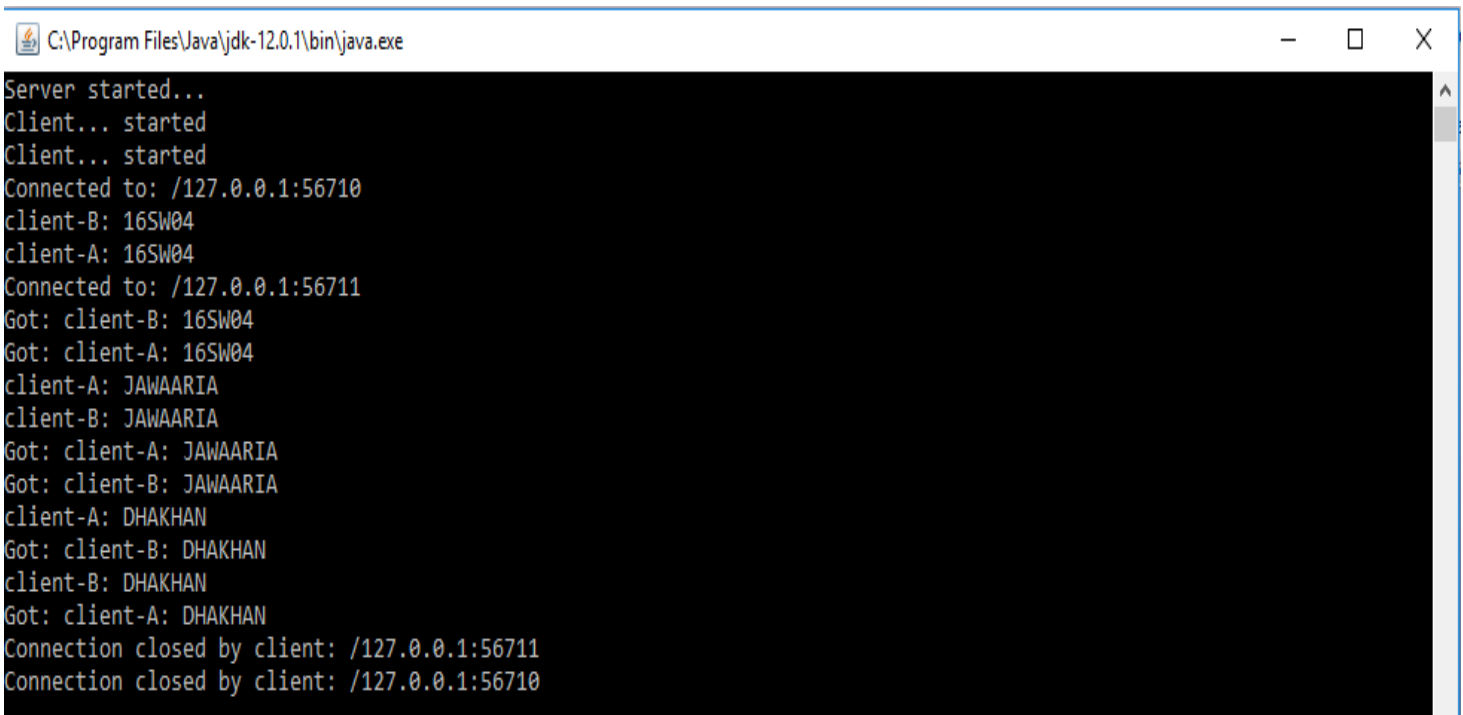
- `ByteBuffer buffer = ByteBuffer.wrap(message);`

and each message is written to the channel from the given buffer on line 25:

- `ByteBuffer buffer = ByteBuffer.wrap(message);`



```
Command Prompt
E:\8th Semester\DC Labs\16SW04\Lab2\NioSocketExample>javac *.java
E:\8th Semester\DC Labs\16SW04\Lab2\NioSocketExample>start java SocketServer
E:\8th Semester\DC Labs\16SW04\Lab2\NioSocketExample>
```



```
C:\Program Files\Java\jdk-12.0.1\bin\java.exe
Server started...
Client... started
Client... started
Connected to: /127.0.0.1:56710
client-B: 16SW04
client-A: 16SW04
Connected to: /127.0.0.1:56711
Got: client-B: 16SW04
Got: client-A: 16SW04
client-A: JAWAARIA
client-B: JAWAARIA
Got: client-A: JAWAARIA
Got: client-B: JAWAARIA
client-A: DHAKHAN
Got: client-B: DHAKHAN
client-B: DHAKHAN
Got: client-A: DHAKHAN
Connection closed by client: /127.0.0.1:56711
Connection closed by client: /127.0.0.1:56710
```