CS4102: Network Programming

Mohamed R. Ghetas

E-mail: Mohghattas@gmail.com

Last update: 16-Nov.-2017

java.net Package

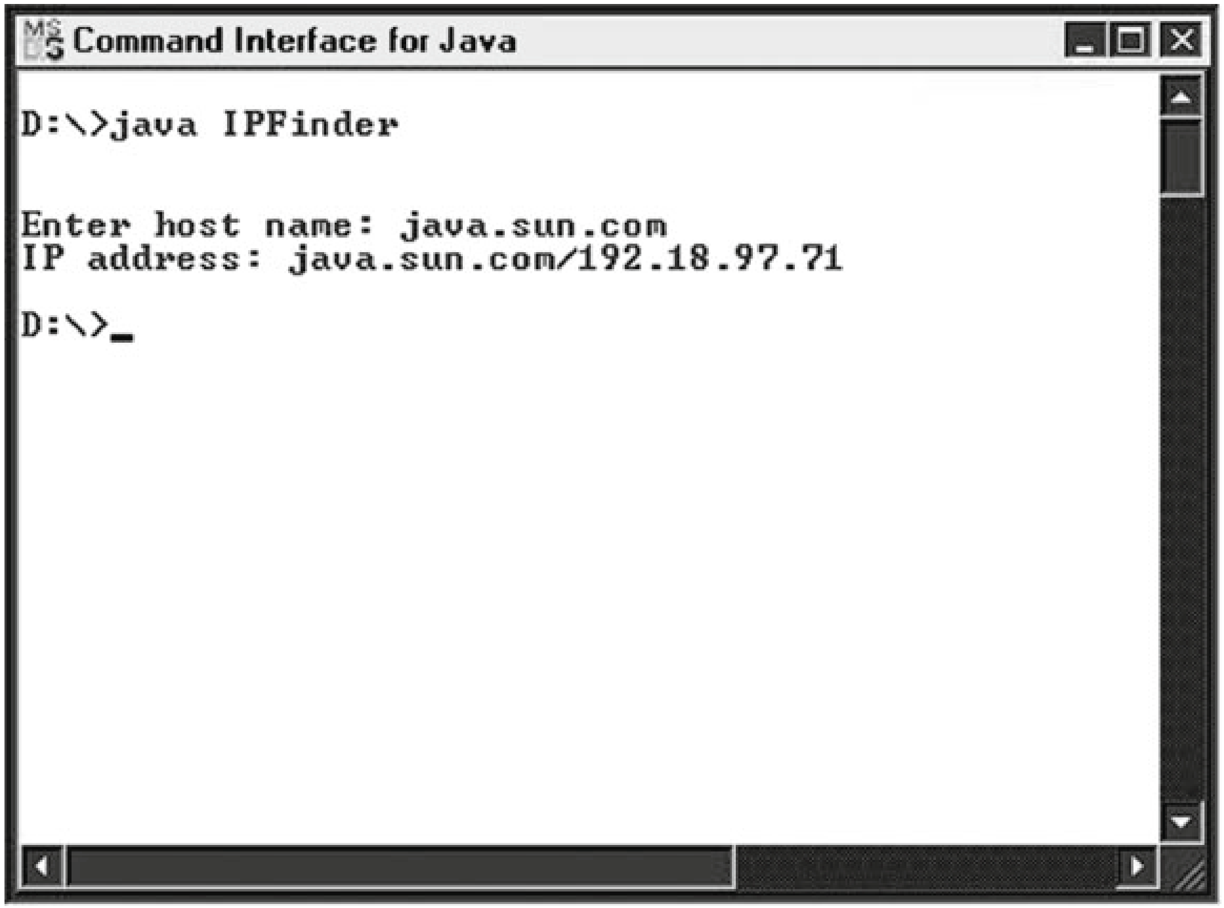
- The Core **java.net** package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing programmers to carry out network programming very easily.
- By using classes from this package, the network programmer can communicate with any server on the Internet or implement his/her own Internet server.
- Full details on them can be found at Oracles website (Java application programming interface (API) full list) `http://docs.oracle.com/javase/7/docs/api/`.

The InetAddress Class

- One of the classes within package **java.net** is called **InetAddress** , which handles Internet addresses both as **host names** and as **IP addresses**.
- Static method **getByName** of this class uses DNS to **return the Internet address** of a specified host name as an InetAddress object. Note it can also handle string representation of IP address.
- In order to display the IP address from this object, we can simply use method println (and use objects **toString** method to be executed).
- **getByName** method throws the checked exception *UnknownHostException* if the host name is not recognized. Thus, one may through this exception or (preferably) handle it with a catch clause.

Example 1

```java
import java.net.*;
import java.util.*;
public class IPFinder
{
public static void main(String[] args)
{
String host;
Scanner input = new Scanner(System.in);
InetAddress address;
System.out.print("\n\nEnter host name: ");
host = input.next();
try
{
address = InetAddress.getByName(host);
System.out.println("IP address: "+ address.toString());
}
catch (UnknownHostException uhEx)
{
System.out.println("Could not find " + host);
}
}
}
```

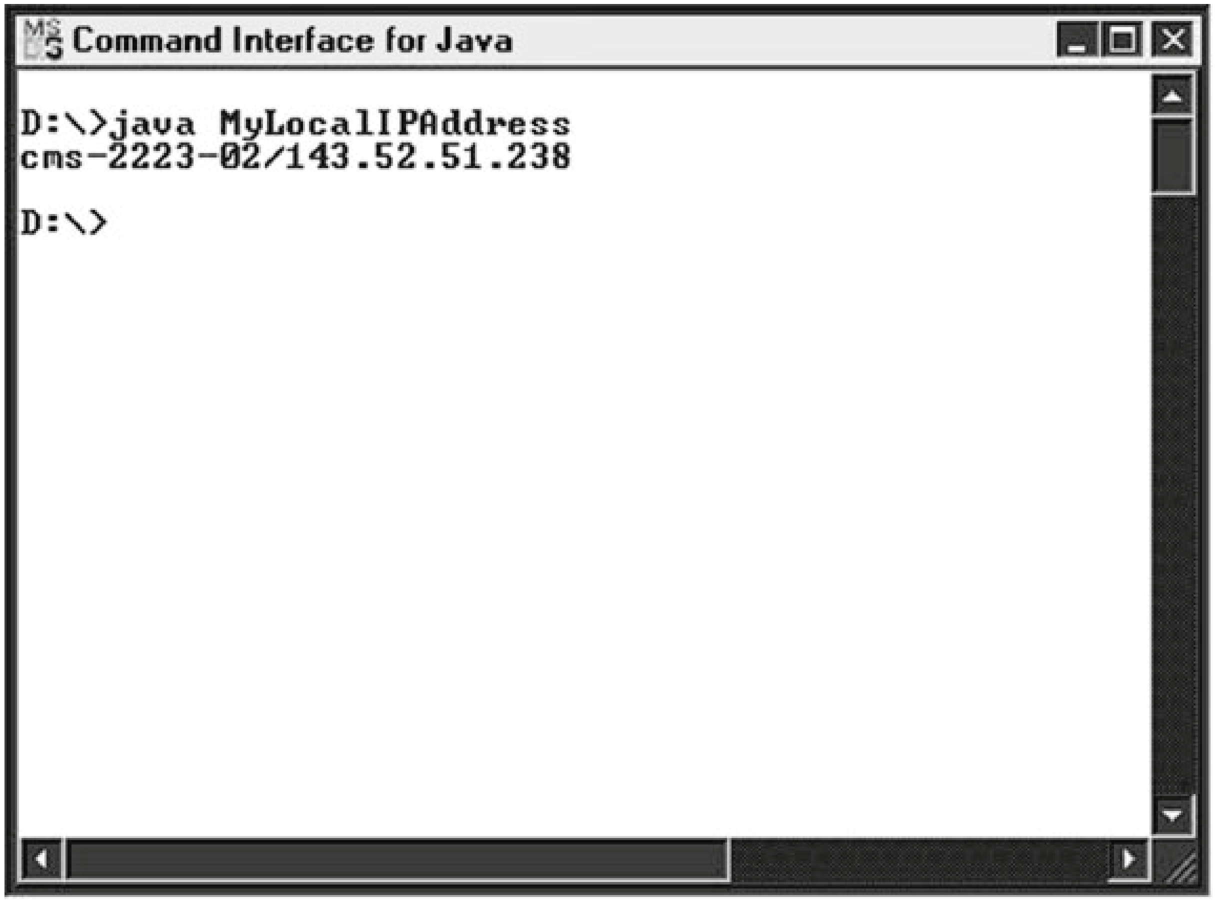


```
D:\>java IPFinder


Enter host name: java.sun.com
IP address: java.sun.com/192.18.97.71

D:\>_
```

Example 2

It is sometimes useful for Java programs to be able to retrieve the IP address of the current machine, as,

```java
import java.net.*;
public class MyLocalIPAddress
{
public static void main(String[] args)
{
try
{
InetAddress address = InetAddress.getLocalHost();
System.out.println(address);
}
catch (UnknownHostException uhEx)
{
System.out.println("Could not find local address!");
}
}
}
```



```
D:\>java MyLocalIPAddress
cms-2223-02/143.52.51.238

D:\>
```

Using Sockets

- Different processes (programs) can communicate with each other across networks by means of sockets.
- Java implements both TCP (TCP/IP) and UDP (datagram) sockets.
- Very often, the two communicating processes will have a client/server relationship.
- The steps required to create client/server programs via each of these methods are very similar.

TCP Sockets

- A communication link created via TCP/IP sockets is a connection-orientated link.
- This means that the connection between server and client remains open throughout the duration of the dialogue between the two and is only broken (under normal circumstances) when one end of the dialogue formally terminates the exchanges (via an agreed protocol).
- Since there are two separate types of process involved (client and server), we shall examine them separately

TCP Sockets:Server process

1. **Create a ServerSocket object:** Requires port number(i.e. $1024-65535$, for non- reserved ones), e.g.

```
ServerSocket myserverSocket = new ServerSocket(1234);
```

   which creates a TCP server socket and binds it to port 1234.

2. **Put the server into a waiting state:** The server waits for a client to connect. It does this by calling method **accept** of class **ServerSocket** , which returns a Socket object when a connection is made,e.g.

```
Socket link = myserverSocket.accept();
```

   which creates another socket for this particular client.

3. **Set up input and output streams:** Methods **getInputStream** and **getOutputStream** of class Socket are used to get references to streams associated with the socket returned in step 2. For a non-GUI application, we can wrap a **Scanner** object around the InputStream object returned by method getInputStream , in order to obtain string-orientated input (just as we would do with input from the standard input stream, System.in), e.g.,

```java
Scanner input = new Scanner(link.getInputStream());
```

   Similarly, we can wrap a **PrintWriter** object around the OutputStream object returned by method getOutputStream . Supplying the PrintWriter constructor with a second argument of true will cause the output buffer to be flushed for every call of println (which is usually desirable), e.g.

```java
PrintWriter output = new PrintWriter(link.getOutputStream(),true);
```

4. **Send and receive data:** Having set up our Scanner and PrintWriter objects, one can simply use method **nextLine** for receiving data and method **println** for sending data, just as we might do for console I/O, i.e.

```java
output.println("Awaiting data ");
String input = input.nextLine();
```

5. **Close the connection (after completion of the dialogue):**, This is achieved via method close of class Socket, e.g.

```java
link.close();
```

TCP Sockets:Server side Example

- In this simple example, the server will accept messages from the client and will keep count of those messages, echoing back each (numbered) message.
- The main protocol for this service is that client and server must alternate between sending and receiving. The dialogue will terminate and final data (if any) should be sent by the server when the client send the string ***CLOSE***.
- **IOException** may be generated by any of the socket operations, so it is preferable to use try and catch blocks.
- It is also good practice to place the closing of the socket in a **finally** clause, so that, whether an exception occurs or not, the socket will be closed (unless, of course, the exception is generated when actually closing the socket, but there is nothing we can do about that). But since the finally clause will need to know about the Socket object (in any case), so it is preferable to declare initial value within its scope for the socket object, in this example it is null.
- Lastly, since the server should handle clients for infinity, the handle client method in this example (initialed when a client make a connection), has been placed inside an infinite loop, e.g.

```
do
{
handleClient();
}while (true);
```

```java
//Server that echoes back client's messages.
//At end of dialogue, sends message indicating
//number of messages received. Uses TCP.
import java.io.*;
import java.net.*;
import java.util.*;
public class TCPEchoServer
{
private static ServerSocket serverSocket;
private static final int PORT = 1234;
public static void main(String[] args)
{
System.out.println("Opening port \n");
try
{
serverSocket = new ServerSocket(PORT); //Step 1.
}
catch(IOException ioEx)
{
System.out.println("Unable to attach to port!");
System.exit(1);
}
do
{
handleClient();
}while (true);
}
private static void handleClient()
{
Socket link = null; //Step 2.
try
```

```java
{
link = serverSocket.accept(); //Step 2.
Scanner input = new Scanner(link.getInputStream()); //Step 3.
PrintWriter output = new PrintWriter(link.getOutputStream(),true); //Step 3.
int numMessages = 0;
String message = input.nextLine(); //Step 4.
while (!message.equals("***CLOSE***"))
{
System.out.println("Message received.");
numMessages++;
output.println("Message " + numMessages+ ": " + message); //Step 4.
message = input.nextLine();
}
output.println(numMessages+ " messages received."); //Step 4.
}
catch(IOException ioEx)
{
ioEx.printStackTrace();
}
finally
{
try
{
System.out.println("\n* Closing connection  *");
link.close(); //Step 5.
}
catch(IOException ioEx)
{
System.out.println("Unable to disconnect!");
System.exit(1);
}
}
}
}
```

TCP Sockets:Client process

Setting up the corresponding client involves four steps:

1. **Establish a connection to the server:** Create a Socket object, supplying its constructor with the **servers IP address** (of type InetAddress) and the **server port number** for the service. For example, assume the server and client sides are running on the same host, so one may create a socket and make a connection by,

```
Socket link = new Socket(InetAddress.getLocalHost(),1234);
```

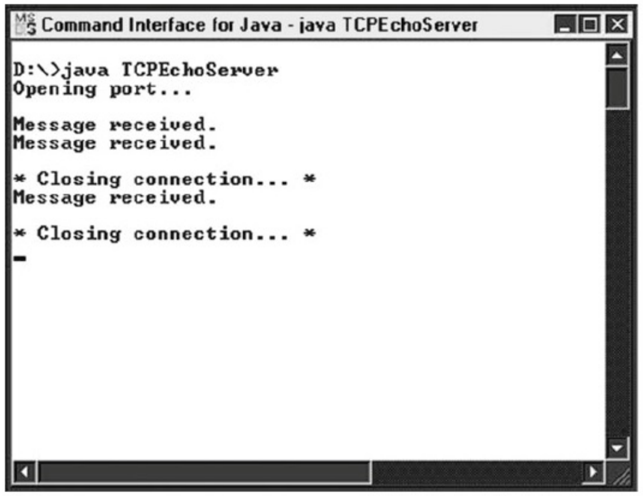
2. **Set up input and output streams:** Exactly like the server side, i.e. by using methods **getInputStream** and **getOutputStream**.
3. **Send and receive data:** Like the server side, i.e. by using **Scanner** and **PrintWriter** objects and their methods methods **nextLine** and **println** respectively.
4. **Close the connection:** Like the server side, i.e. by using method **close** of class **Socket**.

TCP Sockets:Client side Example

```java
import java.io.*;
import java.net.*;
import java.util.*;
public class TCPEchoClient
{
private static InetAddress host;
private static final int PORT = 1234;
public static void main(String[] args)
{
try
{
host = InetAddress.getLocalHost();
}
catch(UnknownHostException uhEx)
{
System.out.println("Host ID not found!");
System.exit(1);
}
accessServer();
}
private static void accessServer()
{
Socket link = null; //Step 1.
try
{
link = new Socket(host,PORT); //Step 1.
Scanner input = new Scanner(link.getInputStream());
//Step 2.
PrintWriter output = new PrintWriter(link.getOutputStream(),true); //Step 2.
//Set up stream for keyboard entry
Scanner userEntry = new Scanner(System.in);
String message, response;
do
{
System.out.print("Enter message: ");
message = userEntry.nextLine();
output.println(message); //Step 3.
response = input.nextLine(); //Step 3.
System.out.println("\nSERVER> "+response);
}while (!message.equals("***CLOSE***"));
}
catch(IOException ioEx)
{
ioEx.printStackTrace();
}
finally
{
try
{
System.out.println("\n* Closing connection  *");
link.close(); //Step 4.
}
catch(IOException ioEx)
{
System.out.println("Unable to disconnect!");
System.exit(1);
}
}
}
```

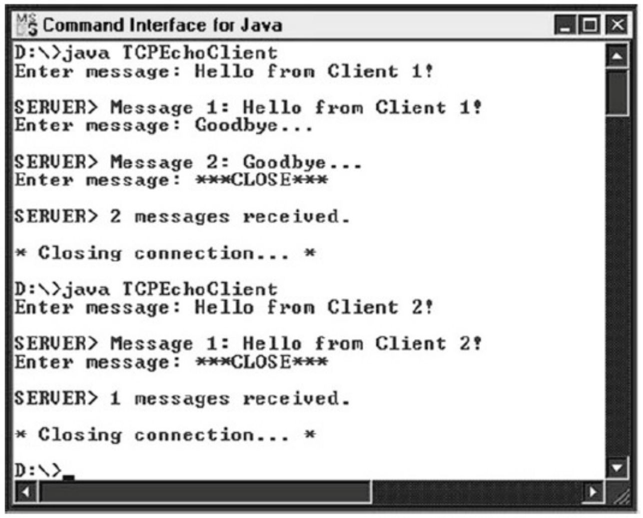
```
}
```

Output screens of running Server and client sides



```
D:\>java TCPEchoServer
Opening port...

Message received.
Message received.

* Closing connection... *
Message received.

* Closing connection... *
```

Example output from the TCPEchoServer program



```
D:\>java TCPEchoClient
Enter message: Hello from Client 1!

SERVER> Message 1: Hello from Client 1!
Enter message: Goodbye...

SERVER> Message 2: Goodbye...
Enter message: ***CLOSE***

SERVER> 2 messages received.

* Closing connection... *

D:\>java TCPEchoClient
Enter message: Hello from Client 2!

SERVER> Message 1: Hello from Client 2!
Enter message: ***CLOSE***

SERVER> 1 messages received.

* Closing connection... *

D:\>_
```

Example output from the TCPEchoClient program

Datagram (UDP) Sockets

- Unlike TCP/IP sockets, datagram sockets are connectionless.
- Since the connection is not maintained between transmissions, the server does not create an individual Socket object for each client, as it did in TCP case.
- Instead of a ServerSocket object, the server creates a **DatagramSocket** object, as does each client when it wants to send datagram(s) to the server.
- The final and most significant difference is that **DatagramPacket** objects are created and sent at both ends, rather than simple strings.

Datagram (UDP) Sockets: Server side

1. **Create a DatagramSocket object:** as in TCP, requires port number, e.g.

```
DatagramSocket mydatagramSocket =new DatagramSocket(1234);
```

   which creates a UDP socket and binds it to port 1234.

2. **Create a buffer for incoming datagrams:** This is achieved by creating an array of bytes, e.g.

```
byte[] buffer = new byte[256];
```

3. **Create a DatagramPacket object for the incoming datagrams:** Requires the previously created byte array and its length, e.g.

```
DatagramPacket inPacket =new DatagramPacket(buffer, buffer.length);
```

4. **Accept an incoming datagram:** Requires the previously defined DatagramPacket object into **receive** method of our **DatagramSocket object**, e.g.

```
mydatagramSocket.receive(inPacket)
```

5. **Accept the senders address and port from the packet:** Methods **getAddress** and **getPort** of our DatagramPacket object are used for this, e.g.

```
InetAddress clientAddress = inPacket.getAddress();
int clientPort = inPacket.getPort();
```

6. **Retrieve the data from the buffer:** For easy handling, the data will be retrieved as a string which requires three arguments, the byte array (by using **getData** method of class DatagramPacket), the start position within the array (typically 0), and the number of bytes (full buffer usually by using **getLength** method of class DatagramPacket). For example,

```java
String message = new String(inPacket.getData(),0,inPacket.getLength());
```

7. **Create the response datagram:** By creating a DatagramPacket object with four arguments, the byte array containing the response message (by the **getBytes** method of the String class), the size of the response, the clients address and the the clients port number. For example,

```java
DatagramPacket outPacket = new DatagramPacket(response.getBytes(), response.
    length(),clientAddress, clientPort);
```

   where response here is a String variable holding the return message.

8. **Send the response datagram:** By calling method **send** of our DatagramSocket object, supplying our outgoing DatagramPacket object as an argument, e.g.

```java
mydatagramSocket.send(outPacket);
```

9. **Close the DatagramSocket (Optional):** Steps $4 - -8$ may be executed indefinitely (within a loop) and the server would probably not be closed down at all. However, if if an exception occurs, then the associated DatagramSocket should be closed by calling method close of our DatagramSocket object, e.g.

```java
mydatagramSocket.close();
```

Datagram (UDP) Sockets: Server side Example

- Similar to the previous example, however, the total number of messages will be the cumulative number of messages (in case there are many clients calling the server).
- For exceptions, the IOException in main is replaced with a SocketException, and there is no checked exception generated by the close method in the finally clause, so there is no try block.

```java
//Server that echoes back client's messages.
//At end of dialogue, sends message indicating number of
//messages received. Uses datagrams.
import java.io.*;
import java.net.*;
public class UDPEchoServer
{
private static final int PORT = 1234;
private static DatagramSocket datagramSocket;
private static DatagramPacket inPacket, outPacket;
private static byte[] buffer;
public static void main(String[] args)
{
System.out.println("Opening port \n");
try
{
datagramSocket =new DatagramSocket(PORT); //Step 1.
}
catch(SocketException sockEx)
{
System.out.println("Unable to open port!");
System.exit(1);
}
handleClient();
}
private static void handleClient()
{
try
{
String messageIn,messageOut;
int numMessages = 0;
InetAddress clientAddress = null;
int clientPort;
do
{
buffer = new byte[256]; //Step 2.
inPacket =new DatagramPacket(buffer, buffer.length); //Step 3.
datagramSocket.receive(inPacket); //Step 4.
clientAddress = inPacket.getAddress(); //Step 5.
clientPort = inPacket.getPort(); //Step 5.
messageIn = new String(inPacket.getData(),0,inPacket.getLength());//Step 6.
System.out.println("Message received.");
numMessages++;
messageOut = "Message " + numMessages+ ": " + messageIn;
outPacket =new DatagramPacket(messageOut.getBytes(),messageOut.length(),
  clientAddress,clientPort); //Step 7.
datagramSocket.send(outPacket); //Step 8.
}while (true);
}
catch(IOException ioEx)
{
ioEx.printStackTrace();
}
finally //If exception thrown, close connection.
{
System.out.println("\n* Closing connection  *");
datagramSocket.close(); //Step 9.
}
}
}
```

Datagram (UDP) Sockets: Client side

Setting up the corresponding client requires the eight steps listed below:

1. **Create a DatagramSocket object:** Similar to the creation of a DatagramSocket object in the server side **but** it doesn't need port number since a default port (at the client end) will be used, e.g.

```java
DatagramSocket mydatagramSocket = new DatagramSocket();
```

2. **Create the outgoing datagram:** Exactly as step 7 of the server side, e.g.

```java
DatagramPacket outPacket = new DatagramPacket(message.getBytes(), message.
   length(), host, PORT);
```

   where host and PORT are the server ones.

3. **Send the datagram message:** As in server side, by calling method **send** of the DatagramSocket object, supplying our outgoing DatagramPacket object as an argument, e.g.

```java
mydatagramSocket.send(outPacket);
```

4. **Create a buffer for incoming datagrams:** Like server, e.g.

```java
byte[] buffer = new byte[256];
```

5. **Create a DatagramPacket object for the incoming datagrams:** Like server, e.g.

```java
DatagramPacket inPacket = new DatagramPacket(buffer, buffer.length);
```

6. **Accept an incoming datagram:** Like server, e.g.,

```java
datagramSocket.receive(inPacket);
```

7. **Retrieve the data from the buffer:** Like server, e.g.

```java
String response = new String(inPacket.getData(),0, inPacket.getLength());
```

8. **Close the DatagramSocket:** Leke server, e.g.

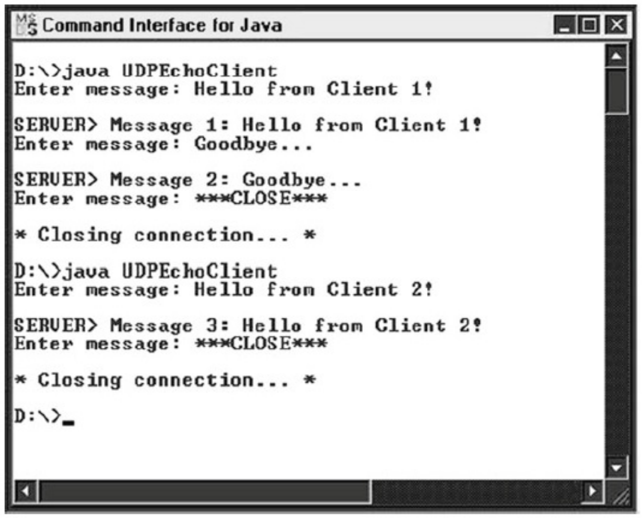
```java
datagramSocket.close();
```

Datagram (UDP) Sockets: Client side Example

```java
import java.io.*;
import java.net.*;
import java.util.*;
public class UDPEchoClient
{
private static InetAddress host;
private static fi nal int PORT = 1234;
private static DatagramSocket datagramSocket;
private static DatagramPacket inPacket, outPacket;
private static byte[] buffer;
public static void main(String[] args)
{
try
{
host = InetAddress.getLocalHost();
}
catch(UnknownHostException uhEx)
{
System.out.println("Host ID not found!");
System.exit(1);
}
accessServer();
}
private static void accessServer()
{
try
{
//Step 1
datagramSocket = new DatagramSocket();
//Set up stream for keyboard entry
Scanner userEntry = new Scanner(System.in);
String message="", response="";
do
{
System.out.print("Enter message: ");
message = userEntry.nextLine();
if (!message.equals("***CLOSE***"))
{
outPacket = new DatagramPacket(
message.getBytes(),
message.length(),
host,PORT);
//Step 2.
//Step 3
datagramSocket.send(outPacket);
buffer = new byte[256]; //Step 4.
inPacket =new DatagramPacket(buffer, buffer.length); //Step 5.
//Step 6
datagramSocket.receive(inPacket);
response =new String(inPacket.getData(),0, inPacket.getLength()); //Step 7.
System.out.println("\nSERVER> "+response);
}
}while (!message.equals("***CLOSE***"));
}
catch(IOException ioEx)
{
ioEx.printStackTrace();
}
finally
```

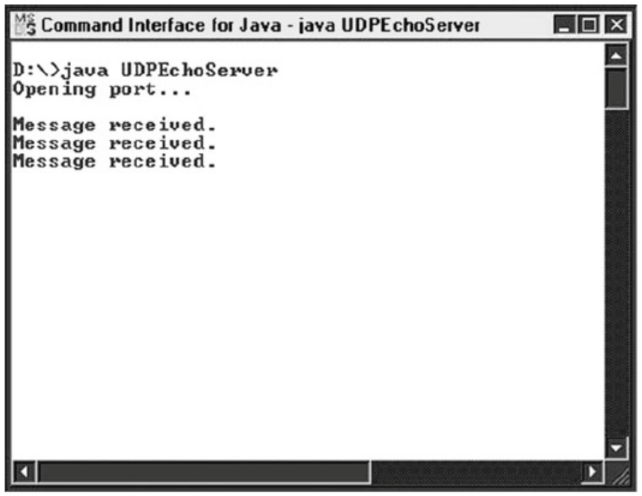
```java
{
System.out.println(
"\n* Closing connection  *");
datagramSocket.close(); //Step 8.
}
}
}
```

Output screens of running Server and client sides



```
D:\>java UDPEchoClient
Enter message: Hello from Client 1!

SERVER> Message 1: Hello from Client 1!
Enter message: Goodbye...

SERVER> Message 2: Goodbye...
Enter message: ***CLOSE***

* Closing connection... *

D:\>java UDPEchoClient
Enter message: Hello from Client 2!

SERVER> Message 3: Hello from Client 2!
Enter message: ***CLOSE***

* Closing connection... *

D:\>_
```

Example output from the UDPEchoClient program (with two clients connecting separately)



```
D:\>java UDPEchoServer
Opening port...

Message received.
Message received.
Message received.
```

Example output from the UDPEchoServer program