CH 2: Starting Network Programming in Java

Contents

2.1 The InetAddress Class

- The class $InetAddress$ handles Internet addresses both as host names and as IP addresses.
  https://docs.oracle.com/javase/7/docs/api/java/net/InetAddress.html
- Example 1: Retrieve the IP address of a given host name

```
Scanner input = new Scanner(System.in);

InetAddress address;
System.out.print("Enter Host Name: ");

String host = input.next();

try{
    address = InetAddress.getByName(host);
    System.out.println("IP address: "+address);
}catch (UnknownHostException ex){
    System.out.println("Couldn't find the host");
}
```

```
Enter Host Name: www.luxor.edu.eg
IP address: www.luxor.edu.eg/193.227.53.75
```

- Static method $getByName$ of this class uses DNS (Domain Name System) to return the Internet address of a specified host name as an $InetAddress$ object.
- The $getByName$ throws the checked exception $UnknownHostException$ if the host name is not recognized.
- Example 2: Retrieve the IP address of the current machine

```
InetAddress address = null;
try {
    address = InetAddress.getLocalHost();
    System.out.println(address);
} catch (UnknownHostException e) {
    System.out.println("Could not find the IP address of
the local host");
}
```

2.2 Using Sockets

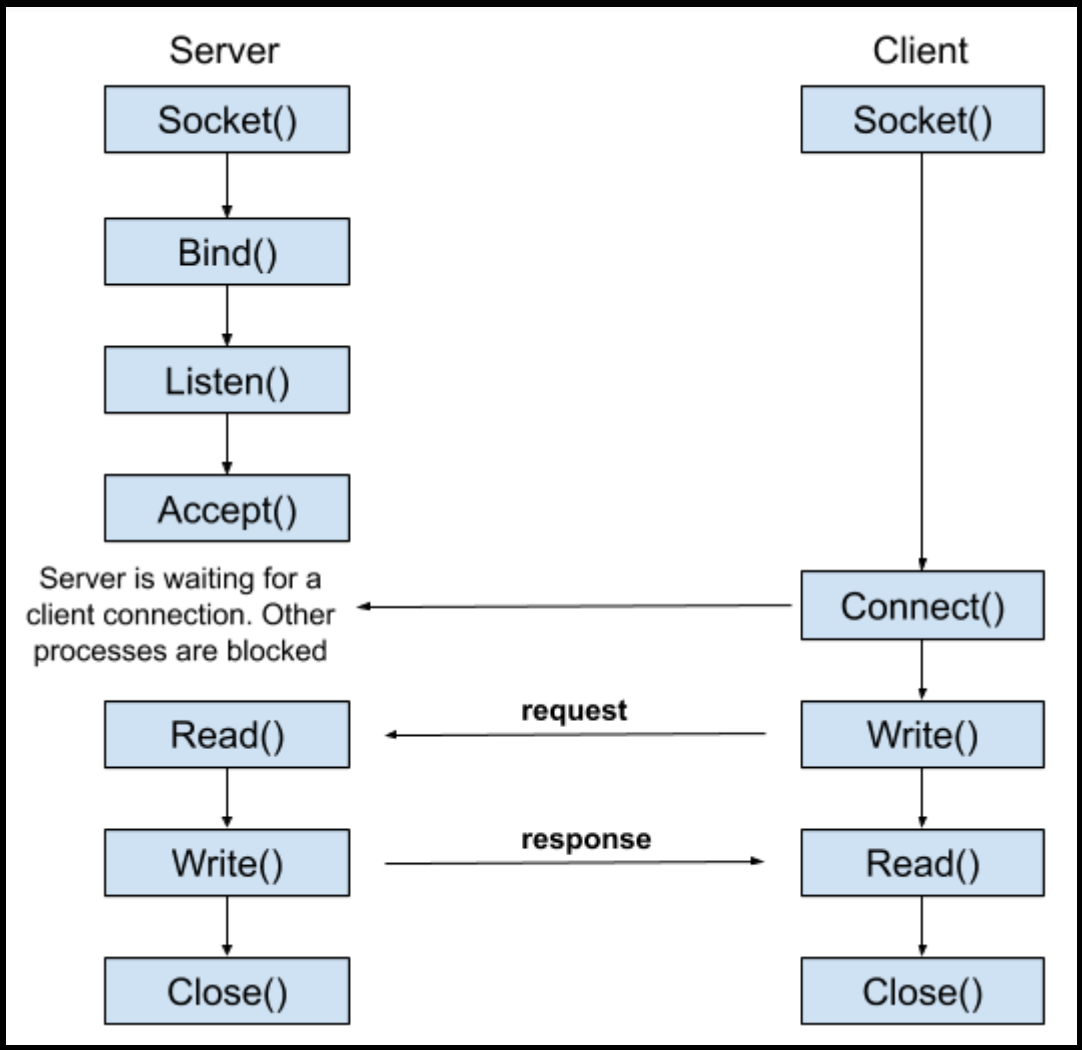
- Java implements both TCP/IP sockets and datagram sockets (UDP sockets).
    1. Both sockets require client/server relationship.

2.2.1 TCP Sockets

- To create a TCP socket for the server:

Step	Description	Code
1. Create a *ServerSocket* object.	The ServerSocket constructor requires a port number (1024–65535)	$ServerSocket\ serverSocket = new\ ServerSocket(1234);$
2. Put the server into a waiting state.	The server waits indefinitely for a client to connect.	$Socket\ link = serverSocket.accept();$
3. Set up input and output streams.	These streams will be used for communication with the client that has just made connection.	$Scanner\ input = new\ Scanner(link.getInputStream());$ $PrintWriter\ output = new\ PrintWriter(link.getOutputStream(), true);$
4. Send and receive data.	We use method nextLine for receiving data and method println for sending data	$output.println("Awaiting\ data\ \ldots");$ $String\ input = input.nextLine();$
5. Close the connection.	This is achieved via method close of class Socket.	$link.close();$

- TCP client/server diagram:



- Example 3: The server will accept messages from the client and will keep count of those messages, echoing back each (numbered) message.

```
public class TCPEchoServer {
    private static ServerSocket serverSocket;
    private static final int PORT = 1234;

    public static void main(String[] args) {
        System.out.println("Opening port ...");
        try {
            serverSocket = new ServerSocket(PORT);
            System.out.println("Port opened successfully...");
        } catch (IOException ioex) {
            System.out.println("Failed to connect to port: " + PORT);
            System.out.println("Please try another port");
            System.exit(1);
        }

        do {
            handleConnection();
        } while (true);
    }

    private static void handleConnection() {
        Socket link = null;

        try {
            link = serverSocket.accept();
            Scanner input = new Scanner(link.getInputStream());
            PrintWriter output = new PrintWriter(link.getOutputStream(), true);

            int num_msgs = 0;
            String msg = input.nextLine();

            while (!msg.equals("***CLOSE***")) {
                System.out.println("Message Received");
                num_msgs++;
                output.println("Message " + num_msgs + ": " + msg);
                msg = input.nextLine();
            }
            output.println(num_msgs + " messages received");
        } catch (IOException ioex) {
            ioex.printStackTrace();
        } finally {
            try {
                System.out.println("\n*Closing Connection...*");
                link.close();
            } catch (IOException ioex) {
                System.out.println("Unable to disconnect!");
                System.exit(1);
            }
        }
    }
}
```

- Step-by-step TCP server code slicing
    1. Create a class named $TCPEchoServer$ with $main()$ method.

```
public class TCPEchoServer {
    public static void main(String[] args) {

    }
}
```

    2. Create two static fields: $ServerSocket$ object for waiting for client requests, and a constant integer $PORT$ to define port number for the service.

```
public class TCPEchoServer {
    private static ServerSocket serverSocket;
    private static final int PORT = 1234;

    public static void main(String[] args) {

    }
}
```

    3. Inside $main$, setup $try - catch$ block

```
try{

}catch (){

}
```

    4. Inside try block, initialize $serverSocket$ object to listen for any incoming connection through port number $PORT$.
    5. Catch $IOException$ inside $catch$ block.
    6. Print appropriate messages for the user.

```
System.out.println("Opening port ...");
try {
    serverSocket = new ServerSocket(PORT);
    System.out.println("Port opened successfully...");
} catch (IOException ioex) {
    System.out.println("Failed to connect to port: " +
PORT);
    System.exit(1);
}
```

7. After $try - catch$ block, define a $do - while$ loop to run the $handleConnection()$ method.

```
public static void main(String[] args) {
    System.out.println("Opening port ...");
    try {
        serverSocket = new ServerSocket(PORT);
        System.out.println("Port opened
successfully...");
    } catch (IOException ioex) {
        System.out.println("Failed to connect to port: "
+ PORT);
        System.out.println("Please try another port");
        System.exit(1);
    }

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

8. Inside the class, define $static$ method for handling incoming connections.

```
public static void main(String[] args) {…}

private static void handleConnection(){

}
```

9. Inside $handleConnection$, define $Socket$ object that will be our link between server and client.
10. Set up $try - catch - finally$ block

```
private static void handleConnection(){
    Socket link = null;

    try{

    }catch (){

    }finally {

    }
}
```

11. Inside *try* block, initialize *link* object to accept incoming connection, create *Scanner* object to get input stream from *link*, and create *PrintWriter* object to set output stream to link.
12. Create an integer variable to count the number of messages.
13. Create a *String* object to receive client message.

```
…
try {
    link = serverSocket.accept();
    Scanner input = new Scanner(link.getInputStream());
    PrintWriter output = new
PrintWriter(link.getOutputStream(), true);

    int num_msgs = 0;
    String msg = input.nextLine();
```

14. Define a *while* loop, with the condition that the incoming message is not equal to **** close ****.
15. Receive a message from the client, increment message counter, send a message to client, and read the incoming message again.
16. If the *while* loop is finished, we print the number of messages received to the server's console.

```
…
try {
    …
while (!msg.equals("***CLOSE***")) {
    System.out.println("Message Received");
    num_msgs++;
    output.println("Message " + num_msgs + ": " + msg);
    msg = input.nextLine();
}
output.println(num_msgs + " messages received");
…
```

17. In the $catch$ block, catch the exception.
18. In the $finally$ block, define a $try - catch$ block to close the link.
19. Print appropriate messages.

```
…
catch (IOException ioex) {
    ioex.printStackTrace();
} finally {
    try {
        System.out.println("\n*Closing
Connection...*\n");
        link.close();
    } catch (IOException ioex) {
        System.out.println("Unable to disconnect!");
        System.exit(1);
    }
}
```

- To create the TCP socket for the client:

Step	Description	Code
1. Establish a connection to the server.	Create a socket object that has the server IP address and the service port number	$Socket\ link = new\ Socket(InetAddress.getLocalHost(), 1234);$
2. Set up input and output streams.	Calling methods getInputStream and getOutputStream of the Socket object	$Scanner\ input = new\ Scanner(link.getInputStream());$ $PrintWriter\ output = new\ PrintWriter(link.getOutputStream(), true);$
3. Send and receive data.	The Scanner object at the client end will receive messages sent by the PrintWriter object at the server end. The PrintWriter object at the client end will send messages that are received by the Scanner object at the server end	$output.println();$ $input.nextLine();$
4. Close the connection.	This is achieved via method close of class Socket.	$link.close();$

- Example 4: The input stream accepts messages from the server, and our client program will need to accept user messages from the keyboard.

```
public class TCPEchoClient {
    private static InetAddress host;
    private static final int PORT = 1234;

    public static void main(String[] args) {
        try {
            host = InetAddress.getLocalHost();
        } catch (IOException ioex) {
            System.out.println("Host ID not found!");
            System.exit(1);
        }
        accessServer();
    }

    private static void accessServer() {
        Socket link = null;
        try{
            link = new Socket(host, PORT);
            Scanner input = new Scanner(link.getInputStream());
            PrintWriter output = new PrintWriter(link.getOutputStream(), true);

            Scanner userInput = new Scanner(System.in);
            String msg, respns;

            do {
                System.out.print("Enter a message: ");
                msg = userInput.nextLine();
                output.println(msg);
                respns = input.nextLine();
                System.out.println("\nSERVER> "+respns);
            }while (!msg.equals("***CLOSE***"));
        } catch (IOException e) {
            e.printStackTrace();
        }finally {
            try {
                System.out.println("CLOSING CONNECTION ...");
                link.close();
            } catch (IOException e) {
                System.out.println("Unable to disconnect");
                System.exit(1);
            }
        }
    }
}
```

- Step-by-Step TCP client code slicing:
    1. Create a $TCPEchoClient$ class with $main$ method.
    2. Define static variables: $InetAddress$ object that represents IP address of the machine, and integer port number $PORT$.
    3. Inside $main$, create $try - catch$ block

```
public class TCPEchoClient {
    private static InetAddress host;
    private static final int PORT = 1234;

    public static void main(String[] args) {
        try {
        } catch () {

        }
    }
}
```

    4. Because our own machine will act as both the client and the server, the server address has the same IP address of the client machine, which is the local host machine. Initialize $host$ object to get local host address, then catch the exception.
    5. Call the method that will access the server.

```
public static void main(String[] args) {
    try {
        host = InetAddress.getLocalHost();
    } catch (IOException ioex) {
        System.out.println("Host ID not found!");
        System.exit(1);
    }
    accessServer();
}
```

6. Create static $accessServer$ method that will be used to connect to TCP server.
7. Within $accessServer$ method, create $Socket$ object that will be our connection link with the server.
8. Create $try - catch - finally$ block.

```
private static void accessServer() {
    Socket link = null;
    try{

    } catch () {

    }finally {

    }
}
```

9. Within try block, initialize $link$ object as a $Socket$ object that gets server address as $host$ variable and service port as $PORT$ variable.
10. Create $Scanner$ object to get input stream from $link$ object, then define $PrintWriter$ object to get output stream from $link$ object.
11. Create another $Scanner$ object that will get user input from client's console.
12. Create two $String$ object, one for sending messages and the other for receiving the message.

```
try{
    link = new Socket(host, PORT);
    Scanner input = new Scanner(link.getInputStream());
    PrintWriter output = new
PrintWriter(link.getOutputStream(), true);

    Scanner userInput = new Scanner(System.in);
    String msg, respns;
…
```

13. Setup a $do - while$ loop in which we accept the user input from console, send the message to the server, accept a response from the server, and print the response on the client's console.
14. The loop runs as long as the user input is not $*** CLOSE ***$.

```
…
do {
    System.out.print("Enter a message: ");
    msg = userInput.nextLine();
    output.println(msg);
    respns = input.nextLine();
    System.out.println("\nSERVER> "+respns);
}while (!msg.equals("***CLOSE***"));
```

15. Catch the exception in $catch$ block.
16. Within the $finally$ block create $try - catch$ block to close the $link$ object.

```
catch (IOException e) {
    e.printStackTrace();
}finally {
    try {
        System.out.println("CLOSING CONNECTION ...");
        link.close();
    } catch (IOException e) {
        System.out.println("Unable to disconnect");
        System.exit(1);
    }
}
```

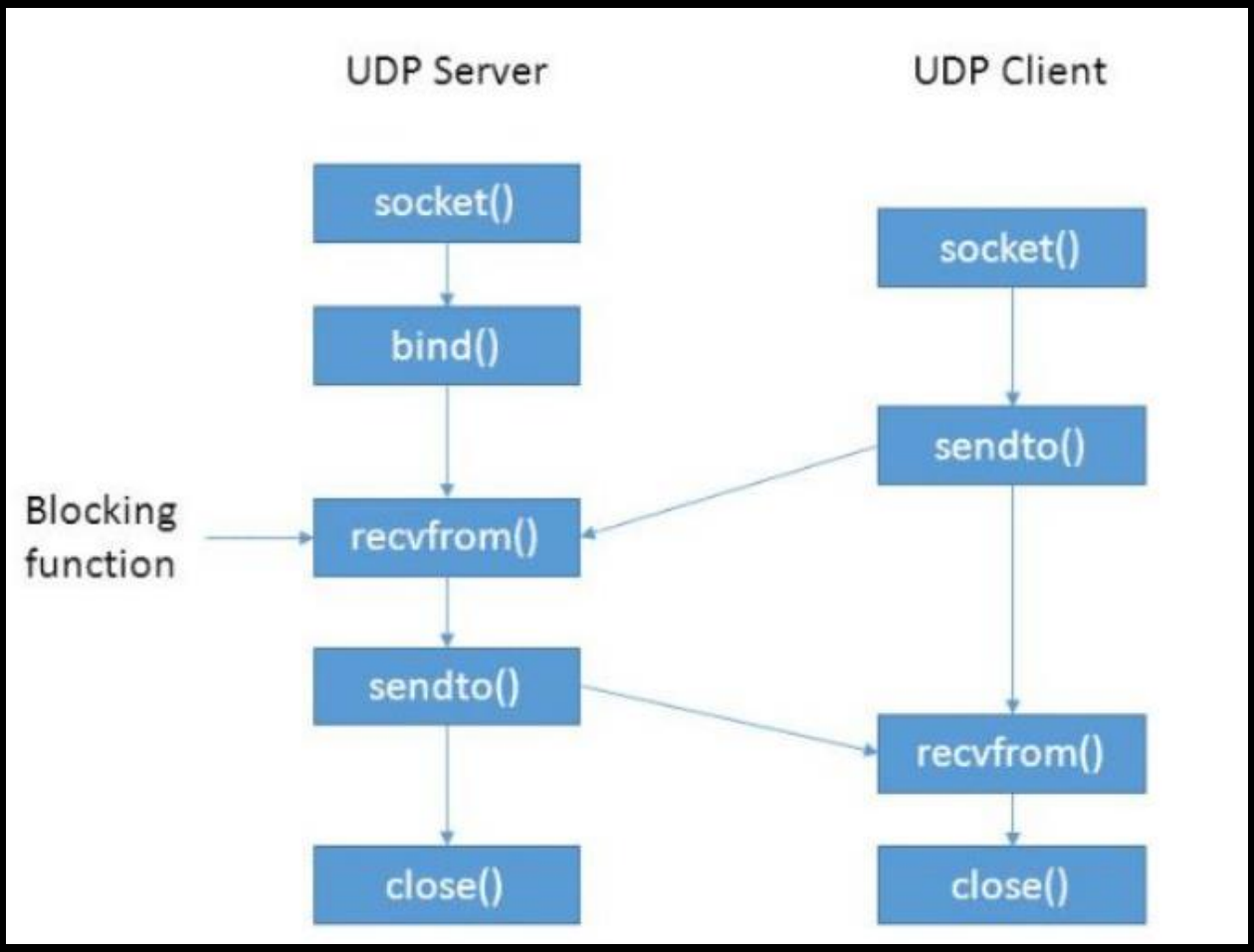
- Now run the server then run the client.

2.2.2 Datagram (UDP) Sockets

- The steps to create a UDP server:

Step	Description	Code
1. Create a *DatagramSocket* object.	supplying the object's constructor with the port number.	$DatagramSocket\ datagramSocket = new\ DatagramSocket(1234);$
2. Create a buffer for incoming datagrams.	This is achieved by creating an array of bytes.	$byte[]\ buffer = new\ byte[256];$
3. Create a DatagramPacket object for the incoming datagrams.	The constructor for this object requires two arguments: the previously-created byte array; and the size of this array.	$DatagramPacket\ inPacket = new\ DatagramPacket(buffer, buffer.length);$
4. Accept an incoming datagram.	This is done via the receive method of our DatagramSocket object	$datagramSocket.receive(inPacket);$
1. Accept the sender's address and port from the packet.	Methods getAddress and getPort of our DatagramPacket object are used for this.	$InetAddress\ clientAddress = inPacket.getAddress();$ $int\ clientPort = inPacket.getPort();$
2. Retrieve the data from the buffer.	the data will be retrieved as a string	$String\ message = new\ String(inPacket.getData(), 0, inPacket.getLength());$
3. Create the response datagram.	Create a DatagramPacket object that recieves responses from client.	$DatagramPacket\ outPacket = new\ DatagramPacket(response.getBytes(), response.length(), clientAddress, clientPort);$
4. Send the response datagram.	Calling method send of our DatagramSocket object, supplying our outgoing DatagramPacket object as an argument.	$datagramSocket.send(outPacket);$
5. Close the DatagramSocket.	Calling method close of our DatagramSocket object.	$datagramSocket.close();$

- UDP client/server diagram:



- Example 5: Create a UDP echo server

```
public class UPDEchoServer {
    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...");

        try { // STEP 1
            datagramSocket = new DatagramSocket(PORT);
            System.out.println("Port opened successfully");
        } catch (IOException ioException) {
            System.out.println("Failed to open the port");
            System.exit(1);
        }
        handleClient();
    }

    private static void handleClient() {
        try {
            String msgIn, msgOut;
            int numMsgs = 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 = datagramSocket.getInetAddress(); //STEP 5
                clientPort = datagramSocket.getPort(); //STEP 5

                msgIn = new String(inPacket.getData(),
                        0, inPacket.getLength()); //STEP 6
                System.out.println("Message Received");
                numMsgs++;

                msgOut = "Message " + numMsgs + ": " + msgIn;
                outPacket = new DatagramPacket(msgOut.getBytes(),
                        msgOut.length(), clientAddress, clientPort); //STEP 7
                datagramSocket.send(outPacket); //STEP 8
            } while (true);
        } catch (IOException ioException) {
            ioException.printStackTrace();
        } finally {
            System.out.println("\n*CLOSING CONNECTION...*");
            datagramSocket.close(); //STEP 9

        }
```

- Step-by-step code slicing:
    1. Create a class $UDPEchoServer$ with $main$ method and a static constant port number $PORT$.

```
public class UDPEchoServer {
    private static final int PORT = 1234;

    public static void main(String[] args) {

    }
}
```

    2. Create $DatagramSocket$ static variable that represents the socket for sending and receiving datagram packets.
    3. Create two static variables of $DatagramPacket$ that represent an incoming datagram packet and an outgoing datagram packet.
    4. Create a static $byte$ array that will receive incoming data.

```
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) {
```

5. In the $main$ method, setup $try - catch$ block.
6. Inside try block, initialize $datagramSocket$ object with port number $PORT$.
7. Catch the exception in the $catch$ block.
8. Call the function $handleClient$.
9. Print appropriate messages to user.

```
public static void main(String[] args) {
    System.out.println("Opening Port...");

    try { // STEP 1
        datagramSocket = new DatagramSocket(PORT);
        System.out.println("Port opened
successfully\n");
    } catch (IOException ioException) {
        System.out.println("Failed to open the port");
        System.exit(1);
    }
    handleClient();
}
```

10. Define static method $handleClient$, and set up $try - catch - finally$ block.

```
private static void handleClient() {
    try {

    } catch () {

    } finally {

    }
}
```

11. Inside try block, define two $String$ objects; one represents incoming message, and the other represents an outgoing message.
12. Inside try block, define an integer for counting messages.
13. Inside try block, define an integer for defining the client port.
14. Inside try block, define an $InetAddress$ object for defining the IP address of the client.

```
try {
    String msgIn, msgOut;
    int numMsgs = 0;
    InetAddress clientAddress = null;
    int clientPort;
…
```

15. Inside try block, set up a $do - while$ loop with the condition $true$. The loop will handle the client connections with the server.

```
do {

} while (true);
```

16. Inside do, initialize the $byte$ array to of size 256 bytes.
17. Inside do, initialize $inPacket$ object that holds the byte array and its length.

```
do {
    buffer = new byte[256]; //STEP 2
    inPacket = new DatagramPacket(buffer,
buffer.length); //STEP 3
```

18. Now the $datagramSocket$ object is ready to receive an incoming packet from the client.
19. From the received packet $inPacket$, extract client' port and IP address.

```
datagramSocket.receive(inPacket); //STEP 4
clientAddress = inPacket.getAddress(); //STEP 5
clientPort = inPacket.getPort(); //STEP 5
```

20. From $inPacket$, we extract the data (client's message) into the string object $msgIn$.
21. Print "message received" to the server console, then increment the counter.

```
msgIn = new String(inPacket.getData(),
        0, inPacket.getLength()); //STEP 6
System.out.println("Message Received");
numMsgs++;
```

22. Define the response of the server that will be sent to client into the String object $msgOut$.
23. Wrap the $msgOut$ object, the message size, client's IP address, and client's port number into the datagram packet $outPacket$.
24. Send the $outPacket$ to the client via $datagramSocket$ object.

```
msgOut = "Message " + numMsgs + ": " + msgIn;
outPacket = new DatagramPacket(msgOut.getBytes(),
        msgOut.length(), clientAddress, clientPort);
//STEP 7
datagramSocket.send(outPacket); //STEP 8
```

25. Catch the exception in $catch$ block.
26. In the $finally$ block, close the connection.

```
catch (IOException ioException) {
    ioException.printStackTrace();
} finally {
    System.out.println("\n*CLOSING CONNECTION...*");
    datagramSocket.close(); //STEP 9
}
```

- The steps to create a UDP client:

Step	Description	Code
1. Create a DatagramSocket object.	The constructor requires no argument, since a default port (at the client end) will be used.	$DatagramSocket\ datagramSocket = new\ DatagramSocket();$
2. Create the outgoing datagram.	Create the packet that will be forwarded to server	$DatagramPacket\ outPacket = new\ DatagramPacket(message.getBytes(), message.length(), host, PORT);$
3. Send the datagram message.	Calling method send of the DatagramSocket object, supplying our outgoing DatagramPacket object as an argument.	$datagramSocket.send(outPacket);$
4. Create a buffer for incoming datagrams.	$byte[]\ buffer = new\ byte[256];$	
5. Create a DatagramPacket object for the incoming datagrams.	$DatagramPacket\ inPacket = new\ DatagramPacket(buffer, buffer.length);$	
6. Accept an incoming datagram.	$datagramSocket.receive(inPacket);$	
7. Retrieve the data from the buffer.	$String\ response = new\ String(inPacket.getData(),0, inPacket.getLength());$	
8. Close the DatagramSocket.	$datagramSocket.close();$	

- Example 6: Create UDP echo client

```
public class UDPEchoClient {
    private static InetAddress host;
    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) {
        try {
            host = InetAddress.getLocalHost();
        } catch (UnknownHostException ex) {
            System.out.println("Host ID not found!");
            System.exit(1);
        }
        accessServer();
    }

    private static void accessServer() {
        try {
            datagramSocket = new DatagramSocket(); //STEP 1
            Scanner userInput = new Scanner(System.in);
            String msg = "", rspns = "";
            do {
                System.out.print("Enter a message: ");
                msg = userInput.nextLine();

                if (!msg.equals("***CLOSE***")) {
                    outPacket = new DatagramPacket(msg.getBytes(),
                            0, msg.length(),
                            host, PORT); //STEP 2
                    datagramSocket.send(outPacket); //STEP 3
                    buffer = new byte[256]; //STEP 4
                    inPacket = new DatagramPacket(buffer, buffer.length);
                    //STEP 5
                    datagramSocket.receive(inPacket); //STEP 6
                    //STEP 7
                    rspns = new String(inPacket.getData(),
                            0, inPacket.getLength());
                    System.out.println("SERVER> " + rspns);
                }
            } while (!msg.equals("***CLOSE***"));
        } catch (IOException ioException) {
            ioException.printStackTrace();
        } finally {
            System.out.println("\n*CLOSING CONNECTION...*");
            datagramSocket.close(); //STEP 8
        }
    }
}
```

- Step-by-step UDP client code slicing:
    1. Create $UDPEchoClient$ class with $main$ method.
    2. Create static variables: $PORT$, $datagramSocket$, $inPacket$, $outPacket$, $buffer$, and $host$.

```
public class UDPEchoClient {
    private static InetAddress host;
    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) {
    }
}
```

    3. Inside $main$, set up $try - catch$ block.
    4. Inside try block, initialize the $host$ object to get the IP address of the local machine. The local machine will act as the server and the client, so both have the same IP address.
    5. Call the $accessServer$ method.

```
public static void main(String[] args) {
    try {
        host = InetAddress.getLocalHost();
    } catch (UnknownHostException ex) {
        System.out.println("Host ID not found!");
        System.exit(1);
    }
    accessServer();
}
```

6. Define static method $accessServer$ that will establish client's connection with the server.
7. Set up the $try - catch - finally$ block.

```
private static void accessServer() {
    try {

    } catch () {

    } finally {

    }
}
```

8. Inside try block, initialize $datagramSocket$ object.
9. Create $Scanner$ object that will get user's input from console.
10. Create two $String$ objects, one for receiving message from server, and the other for sending response to the server.
11. Set up $do - while$ loop, with the condition that the user's input is not "***CLOSE***".

```
try {
    datagramSocket = new DatagramSocket(); //STEP 1
    Scanner userInput = new Scanner(System.in);
    String msg = "", rspns = "";
    do {

        }
    } while (!msg.equals("***CLOSE***"));
```

12. Inside do block, we prompt the user to enter a message, and read it via $userInput$ object.

```
do {
    System.out.print("Enter a message: ");
    msg = userInput.nextLine();
```

13. After reading the user's input, we check that it is not "***CLOSE***" statement. If the user enters "***CLOSE***", the loop terminates and the connection closes.

```
do {
    System.out.print("Enter a message: ");
    msg = userInput.nextLine();
    if (!msg.equals("***CLOSE***")) {

    }
} while (!msg.equals("***CLOSE***"));
```

14. Inside if block, wrap the user's input (message), the message's length, the server's IP address and port into a $outPacket$.
15. Send the $outPacket$ to the server via $datagramSocket$ object.

```
if (!msg.equals("***CLOSE***")) {
    outPacket = new DatagramPacket(msg.getBytes(),
            0, msg.length(),
            host, PORT); //STEP 2
    datagramSocket.send(outPacket); //STEP 3
```

16. Now, the client is ready to receive the response from the server. We initialize our $buffer$ array to receive the server's message.
17. Initialize the $inPacket$ object to hold the $buffer$ and its size.
18. Receive the $inPacket$ from server via $datagramSocket$ object.
19. Extract the $String$ message from $inPacket$, and print the response in the console.

```
buffer = new byte[256]; //STEP 4
inPacket = new DatagramPacket(buffer, buffer.length);
//STEP 5
datagramSocket.receive(inPacket); //STEP 6
//STEP 7
rspns = new String(inPacket.getData(),
        0, inPacket.getLength());
System.out.println("SERVER> " + rspns);
```

20. Catch the exception in the $catch$ block.
21. In the $finally$ block, terminate the connection.
22. Print the appropriate messages to the console.

```
catch (IOException ioException) {
    ioException.printStackTrace();
} finally {
    System.out.println("\n*CLOSING CONNECTION...*");
    datagramSocket.close(); //STEP 8
    System.out.println();
}
```

- Now, run the server then the client

✓ CHECKPOINT: Why we did not define the closing string "***CLOSE***" in the server side in $UDPEchoServer$?

Exercise

1. Create a TCP server that accepts a username and password from a client. If the username and password are valid, the server echoes back user messages. Otherwise, the server prompts the client to enter the username and password again.
2. Summarize the methods of $DatagramSocket$ and $DatagramPacket$ classes.
3. What is $Socket$ class.
4. What is $InetAddress$ class is used for?
5. Write a java application for printing the open ports of the local machine.

❖ Create the same server-client application explained above but with GUI.