Streams, Network Sockets and Threads I – Internet Messenger

Zbigniew Dendzik
Institute of Physics, University of Silesia

2025

Introduction

An important and very frequently used area of Java platform applications is network application programming and, often related to this, multithreaded programming. Today's classes will be devoted to using the most important library classes from the `java.io` and `java.net` packages and basic skills in multithreaded application programming. As you probably remember from the lecture, the basic input-output handling system in Java (package `java.io`) is based on four abstract library classes `InputStream`, `OutputStream`, `Reader` and `Writer` (and corresponding concrete classes, e.g. `BufferedReader`, `PrintWriter`). `InputStream` and `OutputStream` represent binary streams, while `Reader` and `Writer` represent text streams. Today's implementation of a text internet messenger will of course use text streams. Today's exercises also aim to practise exception handling.

Example I

Below you will find the source code of the `Server` and `Client` classes. Create two separate compilation units from them, compile each of them, run (first the server, then the client) and test. If possible, you can also test the example together with a colleague using two computers on a network. Both the client and the server have a static field `PORT` storing the port number (int in the range 0 to 65535 – port numbers from 0 to 1023 are referred to as "well known ports" and reserved for standard services such as www, ssh or email, numbers from 1024 to 49151 are referred to as "registered", and numbers from 49152 to 65535 as "dynamic/private"). The `Client` class also has a static field `HOST` storing the address of the server to which the client will connect.

```
1  import java.io.*;
2  import java.net.*;
3
4  public class Server
5  {
6     public static final int PORT=50007;
7
8     public static void main(String args[]) throws IOException
9     {
10       //creating server socket
```

```
        ServerSocket serv;
        serv=new ServerSocket(PORT);

        //waiting for connection and creating network socket
        System.out.println("Listening: "+serv);
        Socket sock;
        sock=serv.accept();
        System.out.println("Connection established: "+sock);

        //creating data stream from network socket
        BufferedReader inp;
        inp=new BufferedReader(new InputStreamReader(sock.
           getInputStream()));

        //communication - reading data from stream
        String str;
        str=inp.readLine();
        System.out.println("<Received:> " + str);

        //closing connection
        inp.close();
        sock.close();
        serv.close();
    }
}
```

Listing 1: Server.java

```
import java.io.*;
import java.net.*;

public class Client
{
    public static final int PORT=50007;
    public static final String HOST = "127.0.0.1";

    public static void main(String[] args) throws IOException
    {
        //establishing connection with server
        Socket sock;
        sock=new Socket(HOST,PORT);
        System.out.println("Connection established: "+sock);

        //creating data streams from keyboard and to socket
        BufferedReader keyboard;
        keyboard=new BufferedReader(new InputStreamReader(System.in
           ));
        PrintWriter outp;
        outp=new PrintWriter(sock.getOutputStream());

        //communication - reading data from keyboard and sending to
            stream
```

```
23          System.out.print("<Sending:> ");
24          String str=keyboard.readLine();
25          outp.println(str);
26          outp.flush();
27
28          //closing connection
29          keyboard.close();
30          outp.close();
31          sock.close();
32      }
33  }
```

Listing 2: Client.java

Ex. 5.1

Based on the example, write an implementation of an internet messenger that will allow entering successive messages from the keyboard and sending them to the server.

Ex. 5.2

Based on the above example and information from the lecture, write a simple implementation of a text internet messenger operating in simplex mode. Your implementation should include the following stages: (1) establishing connection (this stage has already been done in the example), (2) creating objects representing data streams retrieved from the keyboard and from the network socket and data streams delivered to the network socket, (3) the stage of actual communication in simplex mode (a loop that alternately receives and sends text messages) and (4) termination of communication and (5) closing the connection. Communication should take place until either party sends the message "end" or "END" (the `equalsIgnoreCase()` method of the `String` class may be helpful here). After sending such a message, both the client and the server should print the message "Connection closed" and terminate.

Ex. 5.3

On the client side, identify and handle the exception (`try{}catch()`) that will occur, for example, when the client tries to establish a connection with the server in a situation when no server is running at the specified address and port number. In case of this exception, your messenger should print the message "Connection interrupted" and terminate.

Ex. 5.4

On the client side, handle the `java.net.SocketException` exception (`try/catch`) indicating an error related to the TCP protocol. This exception can be simulated by unplugging the network card socket.

Example II

Below you will find the skeleton implementation of an internet messenger operating in duplex mode (messages can be sent and received in parallel and in any sequence, not necessarily alternately). In this case, sending messages and receiving messages must take place in separate threads.

```
1  import java.io.*;
2  import java.net.*;
3
4  class Receive extends Thread
5  {
6     Socket sock;
7     BufferedReader sockReader;
8
9     public Receive(Socket sock) throws IOException
10    {
11       this.sock=sock;
12       this.sockReader=new BufferedReader(new InputStreamReader(
            sock.getInputStream()));
13    }
14
15    public void run()
16    {
17
18    }
19 }
20
21 public class Server
22 {
23    public static final int PORT=50007;
24
25    public static void main(String args[]) throws IOException
26    {
27       //creating server socket
28       ServerSocket serv;
29       serv=new ServerSocket(PORT);
30
31       //waiting for connection and creating network socket
32       System.out.println("Listening: "+serv);
33       Socket sock;
34       sock=serv.accept();
35       System.out.println("Connection established: "+sock);
36
37       //creating receiving thread
38       new Receive(sock).start();
39
40
41
42       //closing connection
43       serv.close();
44       sock.close();
```

```
45      }
46  }
```

Listing 3: Server.java – duplex mode

```
1   import java.io.*;
2   import java.net.*;
3
4   class Receive extends Thread
5   {
6       Socket sock;
7       BufferedReader sockReader;
8
9       public Receive(Socket sock) throws IOException
10      {
11          this.sock=sock;
12          this.sockReader=new BufferedReader(new InputStreamReader(
                sock.getInputStream()));
13      }
14
15      public void run()
16      {
17
18      }
19  }
20
21  public class Client
22  {
23      public static final int PORT=50007;
24      public static final String HOST = "127.0.0.1";
25
26      public static void main(String[] args) throws IOException
27      {
28          //establishing connection with server
29          Socket sock;
30          sock=new Socket(HOST,PORT);
31          System.out.println("Connection established: "+sock);
32
33          //creating receiving thread
34          new Receive(sock).start();
35
36
37
38          //closing connection
39          sock.close();
40      }
41  }
```

Listing 4: Client.java – duplex mode

Ex. 5.5

Based on the above example, write an implementation of an internet messenger operating in duplex mode (messages can be sent and received in parallel and in any sequence, not necessarily alternately). In this case, sending messages and receiving messages must take place in separate threads. As you probably remember from the lecture, to do this you need to write a class that extends the `java.lang.Thread` library class and place the instructions making up the thread lifecycle in the `public void run()` method of that class. Below you will find the skeleton of such a class. Then in the main thread you need to create an object of this class and call the `start()` method on it. As in the previous example, your messenger should operate until one of the parties sends the message "end" or "END". After sending such a message, both the client and the server should print the message "Connection closed" and terminate.

Example III

Below you will find the skeleton implementation of a multithreaded server that waits for client connections and then creates a separate thread for each client to handle that client.

```
1  import java.io.*;
2  import java.net.*;
3
4  class TaskHandler extends Thread
5  {
6     Socket sock;
7
8     TaskHandler(Socket clientSocket)
9     {
10       this.sock=clientSocket;
11    }
12
13    public void run()
14    {
15
16    }
17 }
18
19 public class Server
20 {
21    public static void main(String[] args) throws IOException
22    {
23       ServerSocket serv=new ServerSocket(80);
24
25       while(true)
26       {
27          //accepting connection
28          System.out.println("Waiting for connection...");
29          Socket sock=serv.accept();
30
31          //creating thread to handle this connection
32          new TaskHandler(sock).start();
```

```
33          }
34      }
35  }
```

Listing 5: Server.java – multithreaded server

Ex. 5.6

Based on the example below, write an implementation of a server that will wait for client requests, accept connections, and then forward messages received from any client to all others. To do this, you should implement a class whose object will be a thread representing the handling of a given client. After establishing a connection with the next client, your server should create a thread that will communicate with that client. You can store objects representing threads using the library implementation of the `java.util.ArrayList` list.