Basic client-server communications

First steps with Java sockets

1. Introduction to sockets

A **socket** is a communication system between processes running on the same or different machines in a network. These processes send and receive information through a connector, and this connector is commonly called *socket*.

1.1. Sockets and port numbers

Each socket has a **port number** associated to it. This number identifies the process that is sending or receiving the information through this socket. So, when a local process wants to communicate with a remote process, they both establish their own port numbers, and whenever the information is sent to each other, the computer knows which process must receive it by checking the destination port.

> For instance, if a machine *A* has a process *PA* listening on port 30, and a machine *B* has another process *PB* listening on port 44, whenever a message gets to machine *B* indicating port number 44, then this message will be received by process *PB* (the machine will redirect the message to that process). On the other side, whenever a message gets to machine *A* indicating port number 30, it will be received by process *PA*.

1.2. Client-server communication

Client-server communication is a concrete type of network communication where:

- One side is set as a **server**, this is, it keeps listening to a given port number, waiting for the other side to send requests or messages.
- The other side is set as a **client**. It must know the server address, and the port number to connect to, and then it can send requests to it.

When a client connects to a server, they establish a socket communication, from which the client sends requests to the server, and the server sends the appropriate responses to the client. For instance, in a web application, we must know the server address (typically its domain name), and its port number (the default port number is 80 for web connections, if we do not specify any). Then, we connect to the server, and we can ask it to send us web pages, documents and other resources.

We have said before that both sides need to have a port number associated to the connection. The port number of the server must be known (for instance, port 80 for web communications, or port 21 for FTP

access), but the client port number is randomly assigned, so that the server will be able to send its responses back to the client.

> For instance, if we try to create a connection to a web server hosted in *www.myserver.com*, we will set port 80 as our destination port. Then, the server will accept our connection, and automatically a random port number (for instance, 9241), will be assigned to our client. From then on, both client and server can send messages to each other to the corresponding port numbers.

Actually, when the connection between client and server is established, the original server port is released to listen for more connections, and another socket is used to communicate with the client. So the client will use the original server port to connect to it, and another (random) port to send data to the server (although client does not need to know this last port when it connects to the server).

1.3. Socket types

There are two basic types of sockets:

- **TCP sockets**, which are connection-oriented sockets. They establish reliable connections, where the delivery of every piece of message is guaranteed. This type of connection is, however, slower than UDP connections that we will see next, because the reception of every data package must be confirmed. So they are used when the integrity of the data being sent is more important than the transfer speed.
    - FTP applications are examples of TCP connections, because the integrity of the files that we download/upload is very important.
- **UDP sockets**, which are non-connection-oriented. Connections are not reliable, and so, there might be some pieces of messages that do not get to its destination. We will use this type of connection when the speed is more important than the integrity of the data.
    - For instance, a live video streaming application uses UDP, because the speed to send and receive the video is more important than losing some frames.

2. Basic usage of Java sockets

In this section we are going to learn how to deal with the two main types of sockets (TCP and UDP) in Java. The core of Java sockets is defined in the `java.net` package. There, we can find some useful classes, as we will see now. You can also download here the source code of the examples that we are going to explain in this section, so you can test them easily.

2.1. Using TCP sockets

If we want to work with TCP sockets, we will need to handle these two classes:

- `ServerSocket` class, that will be used to implement a socket in the server side, that will be listening to client connection requests.
- `Socket` class, that will hold the communication between both sides. We will use a `Socket` object in each side.

In order to connect a client to a server, we will follow these steps:

In the server side:

1. Create a `ServerSocket` object in the server, specifying the desired port number. Regarding port numbers, you must know that ports from 0 to 1023 are usually reserved to other services, such as FTP, HTTP, SMTP... So it is a good idea to choose a port higher than 1023 for our applications.
2. Wait until a client connects, with the accept method.
3. Once the connection is established, we will have a `Socket` object to communicate with that client.
4. Then, we can create input and/or output streams on each side to send or receive data to/from the client.
5. When we finish the communication, we must close everything: input stream, output stream, socket... and server socket, if we don't want to accept more connections. This can be easily done by putting all the closeable objects in the `try` clause (what is called the *try-with-resources* clause).

Let's see all these steps in a piece of code:

```
ServerSocket server = null;
Socket service = null;
DataInputStream socketIn = null;
DataOutputStream socketOut = null;

try (
    ServerSocket server = new ServerSocket(portNumber);
    Socket service = server.accept();
    DataInputStream sIn =
        new DataInputStream(service.getInputStream());
    DataOutputStream sOut =
        new DataOutputStream(service.getOutputStream());
)
{
    ... // Communication process
} catch (IOException e) {
    System.out.println(e);
}
```

In the client side:

1. Create a `Socket` object with the server address and port.
2. When this instruction is completed, we have our connection to the server ready, although it may throw an exception if something goes wrong.
3. Then, we can also create input and/or output streams to send or receive data to/from the server.

If we put all of this together in a piece of code:

```
try (
    Socket mySocket = new Socket("address", portNumber);
    DataInputStream sIn =
        new DataInputStream(mySocket.getInputStream());
    DataOutputStream sOut =
        new DataOutputStream(mySocket.getOutputStream());
)
{
    ... // Communication process
} catch (IOException e) {
    System.out.println(e);
}
```

2.1.1. Example

Let's implement our first, complete example. Whenever we create a client-server application, we normally create two separate projects: one for the server and another one for the client, so that we can distribute both sides of the application separately. In this case, we are going to create a client that says "Hello" to server, and a server that, when receives this message, sends an answer with "Goodbye" to the client.

In the client side, our code should look like this:

```
public class Greet_Client
{
    public static void main(String[] args)
    {
        try (
            Socket mySocket = new Socket("localhost", 2000);
            DataInputStream socketIn =
                new DataInputStream(mySocket.getInputStream());
            DataOutputStream socketOut =
                new DataOutputStream(mySocket.getOutputStream());
        )
        {
            socketOut.writeUTF("Hello");
            String response = socketIn.readUTF();
            System.out.println("Received: " + response);

        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

In the server side, the code is:

```
public class Greet_Server
{
    public static void main(String[] args)
    {
        try (
            ServerSocket server = new ServerSocket(2000);
            Socket service = server.accept();
            DataInputStream socketIn =
                new DataInputStream(service.getInputStream());
            DataOutputStream socketOut =
                new DataOutputStream(service.getOutputStream());
        )
        {
            // Read the message from the client
            String message = socketIn.readUTF();
            // Print the message
            System.out.println("Received: " + message);
            // Answer goodbye to the client
            socketOut.writeUTF("Goodbye");

        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

If we want to run this example (or any other client-server application), we must start by running the server, and when it's waiting for connections, then we run the client. The output should be the text "Received: Hello" in the server console, and the text "Received: Goodbye" in the client console.

Note that the server address for this example has been set to localhost, so that we can test the example in the same computer. But we can change this address to test it with two different machines.

2.1.2. Some implementation issues

If we take a look at previous examples, note that we have used a `DataInputStream` object for reading and a `DataOutputStream` object for writing. We could use other objects for reading or writing, such as `BufferedReader` or `PrintStream`. With these two objects, we could send and receive text messages line by line, instead of using UTF strings.

If the communication between client and server is not synchronous (i.e., the client can send messages to the server at any time and/or there are multiple clients communicating with server at any time), we will need to open the connections (socket, input and/or output streams) inside the try clause (not inside the *try-with-resources* clause), and close them in a finally clause. We will see an example of this when using threads to deal with multiple connections (section 3.2.4).

```
try (ServerSocket server = new ServerSocket(2000))
{
    service = ...
    socketIn = ...
    socketOut = ...
    ...
} catch (IOException e) {
    System.out.println(e);
} finally {
    try {
        if (socketOut != null)
            socketOut.close();
    } catch (IOException ex) {}
    try {
        if (socketIn != null)
            socketIn.close();
    } catch (IOException ex) {}
    try {
        if (service != null)
            service.close();
    } catch (IOException ex) {}
}
```

Also, if we have a graphical application, and we use the sockets or streams from different events, we will not be able to use the *try-with-resources* clause, and we will need to use several *try* clauses to use the socket, or the streams, or close them, in different parts of our application. We will see this in later exercises.

2.1.3. Some useful methods

Regarding the classes that we have just learnt, there are some useful constructors and methods that we may need to have on hand (apart from the ones used in previous examples):

Method(s)	Description
`ServerSocket(int port, int max)`	Creates a server socket with the specified input port and the maximum number of simultaneous connections specified in the 2nd parameter
`Socket(InetAddress addr, int port)`	Creates a socket to connect to the specified InetAddress object and port
`int getLocalPort()`, `int getPort()`	These methods belong to Socket class, and return the port number assigned to this socket in the local machine and remote machine, respectively
`readByte()`, `readChar()`, `readDouble()`, `readFloat()`, `readUTF()`	These methods belong to DataInputStream class, and they can be used to read specific data types from the socket. The last one returns a String in UTF-8 format
`writeBytes(String)`, `writeChars(String)`, `writeDouble(double)`, `writeFloat(float)`, `writeUTF(String)`	These methods belong to DataOutputStream class, and they write specific data types to the socket (the last one writes a String in UTF-8 format)
`print(String message)`, `println(String message)`	These methods belong to PrintStream class, and allows us to write messages to the socket as if we were writing them in the console
`readLine()`	This method belongs to BufferedReader class, and allows us to read messages from the socket line by line, as long as the other side writes them line by line as well

Exercise 1:

Create an "echo" client-server application using Java sockets, by creating these two separate projects (one for the client and another one for the server):

- Create a project for the server side called **Echo_Server**. Define a server that will be listening to port 6000, and when he gets a connection, it will constantly read a message from the client, and send the same message back to it, converted into uppercase. For instance, if it receives the message "Hello", it will return "HELLO" to the client.
- Create a project for the client side called **Echo_Client**. Define a socket that will connect to the server (use "localhost" as server name, if you are running both projects in the same machine). When the connection is set, the client constantly asks the user to enter a message, and then it will send it to the server, waiting for the corresponding echo.
- The communication process will finish when the server receives the message "bye".

2.2. Using UDP sockets

When we use UDP connections, we don't need to establish a previous connection between client and server. Every time that we need to send a message between them, we will need to specify the address and port number of the receiver, and it will get the sender address and port from the message as well. We will work with `DatagramSocket` and `DatagramPacket` classes.

The steps that we need to follow regarding UDP connections are:

1. Create a `DatagramSocket` specifying an IP address and port. Regarding the server, we usually specify a port number known by the client(s), although there are several ways of creating the socket:
   - Connecting to localhost at any available port: `DatagramSocket socket = new DatagramSocket();`
   - Connecting to localhost at a given port number: `DatagramSocket socket = new DatagramSocket(portNumber);`
   - Connecting to a given host and port number: `DatagramSocket socket = new DatagramSocket(portNumber, address);`, where `address` is an `InetAddress` object.
2. As soon as we receive a packet (`DatagramPacket`), we can send messages to the address and port stored in this packet, by using `DatagramPacket` objects as well. We create a `DatagramPacket` object by specifying the message to be sent (as a byte array), its length, and the destination address and port number. Then, we call its `send()` message to send the datagram to the specified destination. To receive it, we create an empty `DatagramPacket` object and call its receive method to fill it with data.

```
String text = "Hello";
byte[] message = text.getBytes();
DatagramPacket packetS = new DatagramPacket(message, message.length,
    InetAddress.getLocalHost(), 2000);
socket.send(packetS);

byte[] buffer = new byte[1024];
DatagramPacket packetR = new DatagramPacket(buffer, buffer.length);
socket.receive(packetR);
```

3. We can also specify a timeout in the socket (in milliseconds), so that if the receive method waits for more than the specified timeout to get a response, then an `InterruptedIOException` is thrown and we can go on.

```
socket.setSoTimeout(2000); // 2 seconds
…
try
{
    …
    socket.receive(packetR);
} catch (InterruptedIOException e) {
    …
}
```

4. When we finish the communication, we must close the socket established. We can also put the socket in the try clause so that it will auto close when the clause finishes.

2.2.1. Example

We are going to implement the same example shown in TCP sockets, but in this case we will use UDP protocol. The client side would be like this:

```
public class GreetUDP_Client
{
    public static void main(String[] args)
    {
        try (DatagramSocket mySocket = new DatagramSocket())
        {
            // Create the packet to be sent
            String text = "Hello";
            byte[] message = text.getBytes();
            DatagramPacket packetS = new DatagramPacket(message, message.length,
                InetAddress.getLocalHost(), 2000);
            mySocket.send(packetS);

            // Receive the response
            byte[] buffer = new byte[1024];
            DatagramPacket packetR = new DatagramPacket(buffer, buffer.length);
            mySocket.receive(packetR);
            System.out.println("Received: " +
                new String(packetR.getData()).trim());
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

And the server side would be like this:

```
public class GreetUDP_Server
{
    public static void main(String[] args)
    {
        try (DatagramSocket mySocket =
            new DatagramSocket(2000, InetAddress.getLocalHost()))
        {
            // Receive the text
            byte[] buffer = new byte[1024];
            DatagramPacket packetR = new DatagramPacket(buffer, buffer.length);
            mySocket.receive(packetR);
            System.out.println("Received: " +
                new String(packetR.getData()).trim());

            // Get host and port from the message
            int destPort = packetR.getPort();
            InetAddress destAddr = packetR.getAddress();

            // Create the response to be sent
            String text = "Goodbye";
            byte[] message = text.getBytes();
            DatagramPacket packetS = new DatagramPacket(message, message.length,
                destAddr, destPort);
            mySocket.send(packetS);

        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

2.2.2. Some implementation issues

As you can see, a datagram is composed of:

- The information or message to be sent
- The message length
- Destination IP and port number
- Local IP and port number (added automatically when creating the datagram)

Note that, when we create a datagram packet to be sent, we convert the message (*String*) into a byte array, with the `getBytes()` method. And when we want to receive a message, we use the `getData()` method to get the bytes of the message, and then we create a *String* with that array and trim it to clean the edges.

Besides, note that, when the server receives the packet from the client, it can retrieve the client host name and port number with the methods `getPort()` and `getAddress()`, since this information is included in the datagram automatically.

> **Exercise 2**:
>
> Create a UDP client-server application with the following projects:
>
> - A project called **UDPDictionary_Client** that will send to the server a word typed by the user. Try to set a timeout in the client (for instance, 5 seconds), by using the `setSoTimeout` method from the socket. If this timeout expires, an `InterruptedIOException` will be thrown, and the client must print a message on the screen with the text "No translation found".
> - A project called **UDPDictionary_Server** that will run on port 6000. It will have a collection (hashtable or something similar) with some words in English (keys) and their corresponding Spanish translation (values). The server will read the word sent by the client, and it will return the Spanish translation of that word. If the word can't be found in the collection, the server will not return anything.

2.3. More about the InetAddress class

You may have read about the `InetAddress` class before, although we have not explained it in detail. It represents an IP address in Java, but it also has methods that connect to a DNS server and resolve a hostname. In other words, we can use an object of this class to connect to a remote server, either by its IP address (not very usual) or by its domain name, that is automatically converted into an IP address.

To create an `InetAddress` object, we must use one of its static factory methods. The most common is:

```
InetAddress address = InetAddress.getByName("www.myserver.com");
```

This method makes a connection to the local DNS server to look up the name and its corresponding IP address. We can also use this method to do a reverse lookup, this is, get the hostname from the IP address:

```
InetAddress address = InetAddress.getByName("201.114.121.65");
```

In both cases, we can check both the hostname and the IP address stored in the InetAddress object by calling the methods `getHostName()` and `getHostAddress()`, respectively.

3. Connecting multiple clients. Sockets and threads

A server that only accepts one client is not a usual server. In real world, a server must be ready to accept multiple clients. To do this, we need to create a thread in the server to deal with each client. So, if we are working with a TCP application, the server code will only create the `ServerSocket` object and go into a loop that creates a socket with each client, and a thread that deals with it:

```
try (ServerSocket server = new ServerSocket(PORT))
{
    System.out.println("Listening...");
    while (true)
    {
        Socket service = server.accept();
        System.out.println("Connection established");
        ServerThread st = new ServerThread(service);
        st.start();
    }
} catch (IOException e) {
    System.out.println(e);
}
```

In the thread, we usually implement all the communication with the client (the input and output streams handling that we did in the server class before):

```
public class ServerThread extends Thread
{
    Socket service;

    public ServerThread(Socket s)
    {
        service = s;
    }

    @Override
    public void run()
    {
        DataInputStream socketIn = null;
        DataOutputStream socketOut = null;
        try
        {
            socketIn = new DataInputStream(service.getInputStream());
            socketOut = new DataOutputStream(service.getOutputStream());

            // … Communication with client

        } catch (IOException e) {
            System.out.println(e);
        } finally {
            try {
                if (socketOut != null)
                    socketOut.close();
            } catch (IOException ex) {}
            try {
                if (socketIn != null)
                    socketIn.close();
            } catch (IOException ex) {}
            try {
                if (service != null)
                    service.close();
            } catch (IOException ex) {}
        }
    }
}
```

Note that, in this case, we can't use the *try* clause to define the socket and input streams inside (i.e. use the *try-with-resources* clause), because the `Socket` object is created in the server main object, and passed to the thread, so it would be closed by the server before the thread could use it. So, we can use a finally clause to close the socket and the input and output streams from the thread.

If we work with UDP servers, we do not need to use any thread, since every datagram sent or received has no relationship with the others, and we do not need to establish a different connection with each client. We only

need to define a loop where the server receives datagrams, gets the remote IP and port number, and sends a response to it.

```
while (true)
{
    byte[] sent = new byte[1024];
    byte[] received = new byte[1024];

    DatagramPacket datagramReceived =
        new DatagramPacket(received, received.length);

    mySocket.receive(datagramReceived);
    ...
    InetAddress remoteIP = datagramReceived.getAddress();
    int remotePort = datagramReceived.getPort();

    DatagramPacket datagramSent = new DatagramPacket (sent,
            sent.length, remoteIP, remotePort);

    mySocket.send(datagramSent);
}
```

Exercise 3:

Improve *Exercise 1* with these two changes:

- Make the client-server connection independent from the machines where they are placed. This is, you must not use "localhost" as the server address. Instead of this, let the user specify the server address.
- Allow more than one client connecting to the server. Then, the server will have to echo the messages from different clients, sending each one its answers.
- Call the new projects **EchoImproved_Server** and **EchoImproved_Client**.