CSE3040 Java Language

Lecture 24: Networking with Java (3)

Dept. of Computer Engineering,
Sogang University

Background: TCP and UDP

- TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are the two main protocols used for communication.
  - Both protocols work on top of IP (Internet Protocol).
- TCP and UDP have different characteristics
  - TCP is used for reliable data transfer
  - UDP is used for applications that prefers fast communication than reliability. Also, UDP is used for multicast and broadcast applications.

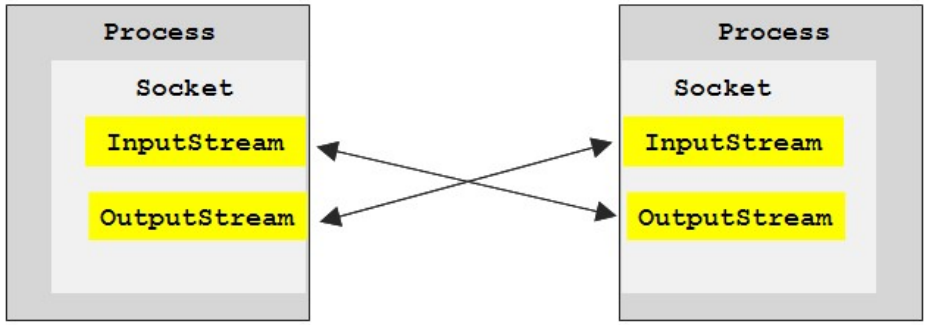
	TCP	UDP
connection type	connection-oriented - a connection should be established before communicating - 1:1 communication	connectionless - no connection established (packets delivered without connection) - 1:1, 1:n, n:n communication
characteristics	- order of data preserved - data delivered reliably - typically slower than UDP	- order of data not preserved - data may be lost - typically faster than TCP
Java classes	Socket ServerSocket	DatagramSocket DatagramPacket MulticastSocket

TCP Socket Programming

- A client and a server communicating through a TCP socket go through the following steps:
  - The server uses ServerSocket to listen on a port and wait for connection requests from clients.
  - The client creates a Socket using server's IP address and port number, and make a connection request to the server.
  - When the ServerSocket receives a client connection request, it creates a new Socket for the client and establish connection between the new Socket and the client Socket.
  - Now, a connection is established and the two Sockets are ready to send data in both ways.

TCP Socket Programming: Socket and ServerSocket

- class Socket
  - manages communication between two processes.
  - has an InputStream and an OutputStream which are used for receiving and sending data.



- class ServerSocket
  - binds with a port and waits for client connection requests.
  - on receiving a connection request, creates a new Socket which will communicate with the client socket.

TCP Socket Programming: Server

- In the server program, a ServerSocket instance is created, which binds with a port.
  - The port number where the server should listen on is given as an argument to the constructor.
- The static method getTime uses library classes Date and SimpleDateFormat to make a String indicating current time.

```
public class Lecture {

    // return current time as a String.
    static String getTime() {
        SimpleDateFormat f = new SimpleDateFormat("[hh:mm:ss]");
        return f.format(new Date());
    }

    public static void main(String args[]) {
        ServerSocket serverSocket = null;
        try {
            // create a ServerSocket instance and bind with port 7777.
            serverSocket = new ServerSocket(7777);
            System.out.println(getTime() + "server is ready.");
        } catch(IOException e) { e.printStackTrace(); }
```

TCP Socket Programming: Server

- When the ServerSocket calls method accept, the method is blocked (does not return) until a client sends connection request.
- When a client connects to the server, method accept returns a Socket instance.
- The server sends a message to the client and closes the socket.
- Then, the server waits for a client again.

```
        while(true) {
            try {
                System.out.println(getTime() + "waiting for clients.");

                Socket socket = serverSocket.accept();
                System.out.println(getTime() + "connection request from " + socket.getInetAddress());

                OutputStream out = socket.getOutputStream();
                DataOutputStream dos = new DataOutputStream(out);

                dos.writeUTF("[Notice] Test Message1 from Server.");
                System.out.println(getTime() + "sent message.");

                dos.close();
                socket.close();
            } catch (IOException e) { e.printStackTrace(); }
        }
    }
}
```

TCP Socket Programming: Client

- The constructor of Socket creates a socket and requests connection to the server.
  - If there is no server waiting on the specified IP address and port number, a ConnectionException occurs.

```
public class Lecture {
    public static void main(String[] args) {
        try {
            String serverIp = "127.0.0.1";  // local address
            System.out.println("connecting to server, IP: " + serverIp);
            Socket socket = new Socket(serverIp, 7777);
            InputStream in = socket.getInputStream();
            DataInputStream dis = new DataInputStream(in);
            System.out.println("message from server: " + dis.readUTF());
            System.out.println("disconnecting...");
            dis.close();
            socket.close();
            System.out.println("disconnected from server.");
        } catch(ConnectException ce) {
            ce.printStackTrace();
        } catch(IOException ie) {
            ie.printStackTrace();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Detour: InputStream & OutputStream

- In File I/O, we have looked at class FileInputStream and FileOutputStream.
- Class FileInputStream is used to read contents from a file.
- Class FileOutputStream is used to write contents to a file.

- Class InputStream is an abstract superclass of FileInputStream, which is a general class used for reading data from a resource. (such as a file, network, etc.)
- Class OutputStream is an abstract superclass of FileOutputStream, which is a general class used for writing data to a resource.

Detour: DataInputStream & DataOutputStream

- DataInputStream and DataOutputStream are classes that provide useful and convenient methods to read data from a stream and write data to a stream.
- Methods defined in DataInputStream

Method	Description
boolean readBoolean()	Reads and returns one input byte.Reads one input byte and returns true if that byte is nonzero, false if that byte is zero.
byte readByte()	Reads and returns one input byte.
char readChar()	Reads two input bytes and returns a char value.
double readDouble()	Reads eight input bytes and returns a double value.
float readFloat()	Reads four input bytes and returns a float value.
void readFully(byte[] b)	Reads some bytes from an input stream and stores them into the buffer array b.
void readFully(byte[] b, int off, int len)	Reads len bytes from an input stream.
int readInt()	Reads four input bytes and returns an int value.
String readLine()	Reads the next line of text from the input stream.
long readLong()	Reads eight input bytes and returns a long value.
short readShort()	Reads two input bytes and returns a short value.
int readUnsignedByte()	Reads one input byte, zero-extends it to type int, and returns the result. (0 through 255)
int readUnsignedShort()	Reads two input bytes and returns an int value in the range 0 through 65535.
String readUTF()	Reads in a string that has been encoded using a modified UTF-8 format.
int skipBytes(int n)	Makes an attempt to skip over n bytes of data from the input stream, discarding the skipped bytes.

Detour: DataInputStream & DataOutputStream

- Methods defined in DataOutputStream

Method	Description
void write(byte[] b)	Writes to the output stream all the bytes in array b.
void write(byte[] b, int off, int len)	Writes len bytes from array b, in order, to the output stream.
void write(int b)	Writes to the output stream the eight low-order bits of the argument b.
void writeBoolean(boolean v)	Writes a boolean value to this output stream.
void writeByte(boolean v)	Writes to the output stream the eight low- order bits of the argument v.
void writeBytes(String s)	Writes a string to the output stream.
void writeChar(int v)	Writes a char value, which is comprised of two bytes, to the output stream.
void writeChars(String s)	Writes every character in the string s, to the output stream, in order, two bytes per character.
void writeDouble(double v)	Writes a double value, which is comprised of eight bytes, to the output stream.
void writeFloat(float v)	Writes a float value, which is comprised of four bytes, to the output stream.
void writeInt(int v)	Writes an int value, which is comprised of four bytes, to the output stream.
void writeLong(long v)	Writes a long value, which is comprised of eight bytes, to the output stream.
void writeShort(int v)	Writes two bytes to the output stream to represent the value of the argument.
void writeUTF(String s)	Writes two bytes of length information to the output stream, followed by the modified UTF-8 representation of every character in the string s.

DataInputStream & DataOutputStream: Example

- Writing to a file using DataOutputStream
  - The result file is not readable as the content is not in a text format.

```
public class Lecture {
    public static void main(String[] args) {
        FileOutputStream fos = null;
        DataOutputStream dos = null;
        try {
            fos = new FileOutputStream("sample.dat");
            dos = new DataOutputStream(fos);
            dos.writeInt(10);
            dos.writeFloat(20.0f);
            dos.writeBoolean(true);
            dos.close();
        } catch (IOException e) { e.printStackTrace(); }
    }
}
```

DataInputStream & DataOutputStream: Example

- Reading from a file using DataInputStream

```
public class Lecture {
    public static void main(String[] args) {
        try {
            FileInputStream fis = new FileInputStream("sample.dat");
            DataInputStream dis = new DataInputStream(fis);
            System.out.println(dis.readInt());
            System.out.println(dis.readFloat());
            System.out.println(dis.readBoolean());
            dis.close();
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

TCP Server using Threads

- To serve multiple clients, the server creates multiple threads.
- To use threads, the class implements Runnable.
- In the constructor, we create a ServerSocket and bind to port 7777.
  - Also, we create an array of threads.

```
public class Lecture implements Runnable {

    ServerSocket serverSocket;
    Thread[] threadArr;

    static String getTime() {
        String name = Thread.currentThread().getName();
        SimpleDateFormat f = new SimpleDateFormat("[hh:mm:ss]");
        return f.format(new Date()) + " " + name + ": ";
    }

    // constructor
    public Lecture(int num) {
        try {
            serverSocket = new ServerSocket(7777);    // create a ServerSocket and bind to port 7777.
            System.out.println(getTime() + "server is ready.");
            threadArr = new Thread[num];
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
```

TCP Server using Threads

- The main method creates a Lecture instance, and calls the start method.
- The start method creates 5 threads and call start for each thread.

```
public static void main(String[] args) {
    Lecture server = new Lecture(5);
    server.start();
}

public void start() {
    for(int i=0; i<threadArr.length; i++) {
        threadArr[i] = new Thread(this);
        threadArr[i].start();
    }
}
```

TCP Server using Threads

- In the run method, the accept method is called to wait for client requests.
- When a client sends a connection request, one of the waiting threads will take the request and continue executing the run method.
  - sends a message and closes the socket.

```
    public void run() {
        while(true) {
            try {
                System.out.println(getTime() + "waiting for clients...");
                Socket socket = serverSocket.accept();
                System.out.println(getTime() + "connection request from " + socket.getInetAddress());
                OutputStream out = socket.getOutputStream();
                DataOutputStream dos = new DataOutputStream(out);
                dos.writeUTF("[Notice] Test Message1 from Server.");
                System.out.println(getTime() + "sent message");
                dos.close();
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Chat Program using TCP Sockets

- ChatServer.java
  - a chat server which listens on port 7777.
  - When a client connects, a Sender object and a Receiver object is created which take care of communicating with the client.

```
public class ChatServer {
    public static void main(String args[]) {
        ServerSocket serverSocket = null;
        Socket socket = null;
        try {
            serverSocket = new ServerSocket(7777);
            System.out.println("server is ready.");
            socket = serverSocket.accept();
            Sender sender = new Sender(socket);
            Receiver receiver = new Receiver(socket);
            sender.start();
            receiver.start();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Chat Program using TCP Sockets

- ChatServer.java (cont.)
  - Sender is a thread that reads user input and sends the input through the socket.

```
class Sender extends Thread {
    Socket socket;
    DataOutputStream out;
    String name;

    Sender(Socket socket) {
        this.socket = socket;
        try {
            out = new DataOutputStream(socket.getOutputStream());
            name = "["+socket.getInetAddress()+":"+socket.getPort()+"]";
        } catch(Exception e) {}
    }
    public void run() {
        Scanner scanner = new Scanner(System.in);
        while (out != null) {
            try {
                out.writeUTF(name+scanner.nextLine());
            } catch(IOException e) {}
        }
    }
}
```

Chat Program using TCP Sockets

- ChatServer.java (cont.)
  - Receiver is a thread that reads data from the socket and prints it on the screen.

```
class Receiver extends Thread {
    Socket socket;
    DataInputStream in;
    Receiver(Socket socket) {
        this.socket = socket;
        try {
            in = new DataInputStream(socket.getInputStream());
        } catch(IOException e) {}
    }

    public void run() {
        while ( in != null ) {
            try {
                System.out.println(in.readUTF());
            } catch(IOException e) {}
        }
    }
}
```

Chat Program using TCP Sockets

- ChatClient.java
  - ChatClient.java uses Sender and Receiver classes that are already defined in ChatServer.java. These two files should be in the same package.

```
public class ChatClient {
    public static void main(String args[]) {
        try {
            String serverIp = "127.0.0.1";
            Socket socket = new Socket(serverIp, 7777);
            System.out.println("connected to server.");
            Sender sender = new Sender(socket);
            Receiver receiver = new Receiver(socket);
            sender.start();
            receiver.start();
        } catch(ConnectException ce) {
            ce.printStackTrace();
        } catch(IOException ie) {
            ie.printStackTrace();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Programming Lab #24

24-01. TCP Server & Client

- The project consists of two files: Server.java and Client.java.
- You need to run two programs
  - If you are using eclipse, use two consoles, one for each program
  - You can run the programs from the command line.
    - You will need to set the classpath to where your .class files are.

- Server.java

```
public class Server {
     // return current time as a String.
    static String getTime() {
        SimpleDateFormat f = new SimpleDateFormat("[hh:mm:ss]");
        return f.format(new Date());
    }

    public static void main(String args[]) {
        ServerSocket serverSocket = null;
        try {
            // create a ServerSocket instance and bind with port 7777.
            serverSocket = new ServerSocket(7777);
            System.out.println(getTime() + "server is ready.");
        } catch(IOException e) { e.printStackTrace(); }
```

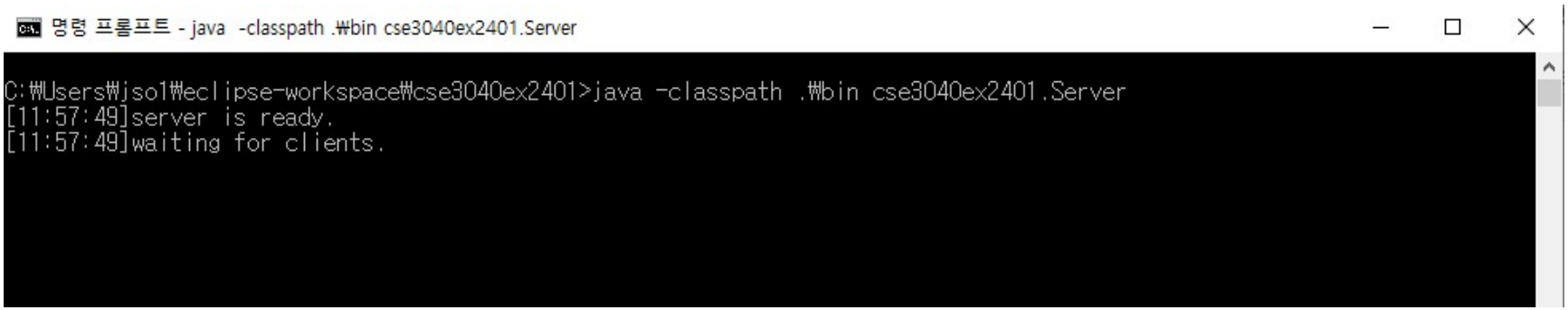
24-01. TCP Server & Client

- Server.java (cont.)

```
        while(true) {
            try {
                System.out.println(getTime() + "waiting for clients.");

                Socket socket = serverSocket.accept();
                System.out.println(getTime() + "connection request from " + socket.getInetAddress());

                OutputStream out = socket.getOutputStream();
                DataOutputStream dos = new DataOutputStream(out);

                dos.writeUTF("[Notice] Test Message1 from Server.");
                System.out.println(getTime() + "sent message.");

                dos.close();
                socket.close();
            } catch (IOException e) { e.printStackTrace(); }
        }
    }
}
```

24-01. TCP Server & Client

- Running programs from command prompt shell (cmd)
  - Suppose your workspace directory is "C:\Users\jso1\eclipse-workspace".
  - If your project name is "cse3040ex2401", your files will be under:
    - C:\Users\jso1\eclipse-workspace\cse3040ex2401.
  - Suppose your package name is "cse3040ex2401". Then, from the project directory,
    - Your source (.java) files are in ".\src\cse3040ex2401".
    - Your byte code (.class) files are in ".\bin\cse3040ex2401", if you compiled the source files from eclipse.
  - To run the server program, you should use the following command from your project directory
    - `java -classpath .\bin cse3040ex2401.Server`



24-02. DataWriter & DataReader

- The project consists of two files: DataWriter.java and DataReader.java.
- Execute DataWriter first, and then DataReader. Understand the results.

- DataWriter.java

```
public class DataWriter {
     public static void main(String[] args) {
        FileOutputStream fos = null;
        DataOutputStream dos = null;
        try {
            fos = new FileOutputStream("sample.dat");
            dos = new DataOutputStream(fos);
            dos.writeInt(10);
            dos.writeFloat(20.0f);
            dos.writeBoolean(true);
            dos.close();
        } catch (IOException e) { e.printStackTrace(); }
    }
}
```

24-02. DataWriter & DataReader

- The project consists of two files: DataWriter.java and DataReader.java.
- Execute DataWriter first, and then DataReader. Understand the results.

- DataReader.java

```
public class DataReader {
     public static void main(String[] args) {
        try {
            FileInputStream fis = new FileInputStream("sample.dat");
            DataInputStream dis = new DataInputStream(fis);
            System.out.println(dis.readInt());
            System.out.println(dis.readFloat());
            System.out.println(dis.readBoolean());
            dis.close();
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

24-03. Server using multiple threads

- Run this server, and use the client from problem 24-01 to connect to the server.

```
public class Server implements Runnable {

    ServerSocket serverSocket;
    Thread[] threadArr;

    static String getTime() {
        String name = Thread.currentThread().getName();
        SimpleDateFormat f = new SimpleDateFormat("[hh:mm:ss]");
        return f.format(new Date()) + " " + name + ": ";
    }

    // constructor
    public Server(int num) {
        try {
            serverSocket = new ServerSocket(7777);    // create a ServerSocket and bind to port 7777.
            System.out.println(getTime() + "server is ready.");
            threadArr = new Thread[num];
        } catch(IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Server server = new Server(5);
        server.start();
    }
```

24-03. Server using multiple threads

- Run this server, and use the client from problem 24-01 to connect to the server. (cont.)

```
 public void start() {
        for(int i=0; i<threadArr.length; i++) {
            threadArr[i] = new Thread(this);
            threadArr[i].start();
        }
    }

    public void run() {
        while(true) {
            try {
                System.out.println(getTime() + "waiting for clients...");
                Socket socket = serverSocket.accept();
                System.out.println(getTime() + "connection request from " + socket.getInetAddress());
                OutputStream out = socket.getOutputStream();
                DataOutputStream dos = new DataOutputStream(out);
                dos.writeUTF("[Notice] Test Message1 from Server.");
                System.out.println(getTime() + "sent message");
                dos.close();
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

24-04. Chat server & Chat client

- The project consists of two files: ChatServer.java and ChatClient.java.
- Execute ChatServer first and then ChatClient. Understand the results.

- ChatServer.java

```
public class ChatServer {
    public static void main(String args[]) {
        ServerSocket serverSocket = null;
        Socket socket = null;
        try {
            serverSocket = new ServerSocket(7777);
            System.out.println("server is ready.");
            socket = serverSocket.accept();
            Sender sender = new Sender(socket);
            Receiver receiver = new Receiver(socket);
            sender.start();
            receiver.start();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

24-04. Chat server & Chat client

- The project consists of two files: ChatServer.java and ChatClient.java.
- Execute ChatServer first and then ChatClient. Understand the results.

- ChatServer.java (cont.)

```
class Sender extends Thread {
    Socket socket;
    DataOutputStream out;
    String name;

    Sender(Socket socket) {
        this.socket = socket;
        try {
            out = new DataOutputStream(socket.getOutputStream());
            name = "["+socket.getInetAddress()+":"+socket.getPort()+"]";
        } catch(Exception e) {}
    }
    public void run() {
        Scanner scanner = new Scanner(System.in);
        while (out != null) {
            try {
                out.writeUTF(name+scanner.nextLine());
            } catch(IOException e) {}
        }
    }
}
```

24-04. Chat server & Chat client

- The project consists of two files: ChatServer.java and ChatClient.java.
- Execute ChatServer first and then ChatClient. Understand the results.

- ChatServer.java (cont.)

```
class Receiver extends Thread {
    Socket socket;
    DataInputStream in;
    Receiver(Socket socket) {
        this.socket = socket;
        try {
            in = new DataInputStream(socket.getInputStream());
        } catch(IOException e) {}
    }

    public void run() {
        while ( in != null ) {
            try {
                System.out.println(in.readUTF());
            } catch(IOException e) {}
        }
    }
}
```

24-04. Chat server & Chat client

- The project consists of two files: ChatServer.java and ChatClient.java.
- Execute ChatServer first and then ChatClient. Understand the results.

- ChatClient.java
  - ChatClient uses class Sender and class Receiver, defined in ChatServer.java.

```
public class ChatClient {
    public static void main(String args[]) {
        try {
            String serverIp = "127.0.0.1";
            Socket socket = new Socket(serverIp, 7777);
            System.out.println("connected to server.");
            Sender sender = new Sender(socket);
            Receiver receiver = new Receiver(socket);
            sender.start();
            receiver.start();
        } catch(ConnectException ce) {
            ce.printStackTrace();
        } catch(IOException ie) {
            ie.printStackTrace();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

End of Class

Instructor office: AS818A

Email: jso1@sogang.ac.kr