# Comunicarea in retea

## Introducere

In acest laborator vom vorbi despre comunicarea intre doua sisteme. Unul din aceste sisteme (sau ambele) va fi implementat in Java.
Pentru ca doua entitati sa comunice, in general, trebuie sa avem un initiator al comunicarii, si un receptor, care raspunde initiatorului. Imaginati-va ca suntem pe strada si vedem un individ A care abordeaza pe un alt individ B si il intreaba ceva. A este initiator, iar cand vorbim de sisteme, spunem ca A este CLIENT. B este cel care raspunde la comunicare si spunem ca este SERVER.
Trebuie remarcat ca daca vedem momentul in care cei doi incep comunicarea (A si B) putem spune cine a initiat si cine a raspuns, insa daca observam dialogul fara sa vedem inceputul, nu mai stim acest lucru. Similar, cand doua sisteme comunica, rolul de server si cel de client nu este relevant odata ce comunicarea a fost stabilita, amandoi partenerii fiind egali.

Orice fel de operatie de I/O in java se face prin streamuri (Input si Output Stream), deci si comunicarea pe retea. Astfel, pentru a putea comunica, trebuie sa obtinem un OutputStream, pentru a trimite informatie, si un InputStream pentru a primi informatie. Aceste streamuri, in cazul comunicarii pe retea se obtin dintr-un obiect Socket, obiect care reprezinta o conexiune cu un computer partener. Cu alte cuvinte, scopul nostru este de a obtine un obiect Socket.

## Configurarea unui server

Asa cum am spus serverul este cel care asteapta sa fie abordat de client. Comunicarea in Java se face (varianta standard) se face prin protocolul TCP/IP, iar pentru a “porni” un server trebuie sa intruim sistemul sa “asculte” pe o anumita adresa IP si pe un anumit port.
Pentru a configura un server, trebuie doar sa instantiem un obiect de tipul ServerSocket:

```
ServerSocket ss = new ServerSocket(port);
```

unde “port” este un port pe care se asulta. Se poate folosi si un alt constructor, in care este specificat pe al doilea parametru o adresa IP pe care sa aculte serverul (altfel se asulta pe toate interfetele).
Dupa ce facem acest lucru trebuie sa ii spunem serverului sa astepte un client. Acest lucru il facem chemand metoda accept pe obiectul server socket.

```
Socket socket = ss.accept();
```

Metoda “accept” este bocanta, pana cand un client se conecteaza (sau apare un timeout sau o eroare). Dupa ce un client se conecteaza, este returnat obiectul Socket corespunzator conexiunii si din acest moment comunicarea poate incepe. Voi explica ulterior cum se foloseste obiectul socket.

## Configurarea unui client

Clientul este lansat mult mai simplu. Pur si simplu instantiem un obiect de tip Socket:

```
Socket socket = new Socket(ipAddressOrHostName, port);
```
Observam ca de data aceasta trebuie specificata atat adresa IP a serverului cat si portul pe care se va realiza conexiunea (portul destinatie). Daca conexiunea are succes, atunci continuam normal. Daca este aruncata o exceptie, inseamna ca nu s-a putut realiza conexiunea.

## Folosirea unui Socket

Dupa ce am obtinut un Socket, nu mai conteaza cu cine avem de-a face, client sau server, lucrurile sunt absolut identice.
Primul pas este sa obtinem un OutputStream:
```
OutputStream os = socket.getOutputStream();
```
urmat de InputStream:
```
InputStream is = socket.getInputStream();
```
nu uitam ca acestea doua sunt LOW level streams, deci ele pot fi folosite pentru a putea construi high level streams (cum ar fi ObjectOutputStream si ObjectInputStream) care ne permit o comunicare lejera pe retea.
Cateodata sisteul de operare poate alege sa nu trimita imediat datele pe retea, pentru a fragmenta comunicarea cat mai putin. Daca totusi dorim sa trimitem imediat mesajul, putem folosi metoda “flush” fara niciun parametru de pe OutputStream.
Mare atentie, ca orice resursa de IO, InputStream, OutputStream si Socket trebuie inchise!

## Lucrul in paralel

Atunci cand avem de-a face cu apeluri blocante, cum ar fi serverSocket.accept() sau, dupa ce obtinem un ObjectInputStream dintr-un socket, objectInputStream.readObject() nu este de dorit sa avem comunicarea blocanta, si sa ne opreasca interfata grafica sau alte procese. Pentru asta, trebuie sa avem grija sa lansam aceste apeluri pe un fir de executie paralel.
Lansarea firelor de executie in Java este un proces foarte simplu, din 3 pasi.
Primul pas este sa instantiem un obiect de un tip care implementeaza interfata Runnable.
```
Runnable runnable = new Runnable() {
public void run()
{
/// stuff to do
}
}
```
Apoi sa instantiem un obiect de tip Thread folosind acest obiect Runnable in constructorul sau:
```
Thread thread = new Thread(runnable);
```
Si in fine sa lansam in executie firul:
```
thread.start();
```
ATENTIE! Este gresit sa chemam metoda thread.run()! Aceasta doar executa pe loc codul din Runnable, dar in threadul curent!

Din momentul in care chemam thread.start(), in programul nostru vor exista doua fire de executie care ruleaza in paralel.

Lucrul in paralel poate duce la probleme de concurenta, atunci cand doua threaduri incearca sa acceseze si sa modifice aceeasi zona de memorie (acelasi state). Pentru a elimina aceste probleme, putem sa fortam un anumit set de operatii sa fie “atomice” cu ajutorul lui cuvantului cheie “synchronized”.
Accesul intr-un bloc synchronized este permis unui singur thread, primul care reuseste sa ocupe semaforul pe care s-a facut sincronizarea.
Pentru metodele statice, semaforul este obiectul Class pentru clasa respectiva.
Pentru metodele ne-statice, semaforul este referinta la this pentru instanta corespunzatoare metodei.
Pentru blocurile synchronized declarate explicit, semaforul este cel care apare intre parantezele lui synchronized.
Atentie! Obiectul pe care se face sincronizarea trebie sa nu fie null!
Daca un thread ajunge la un synchronized iar monitorul este deja ocupa de un alt Thread, atunci Threadul curent asteapta eliberarea monitorului.

## Exercitiul 1

Sa se defineasca clasa “Mesaj” cu campurile:

- Date dataMesaj;
- String mesaj;
- String sender;

Scrieti un server care sa asculte conexiuni noi pe portul 12345.
Serverul va pastra o lista interna de Mesaj.
Dupa ce un client va fi conectat, serverul va trimite mai intai toate obiectele de tip Mesaj din lista interna si apoi va citi obiecte de tip Mesaj de la client, pe care le va salva in lista interna.

## Exercitiul 2

Scrieti un client care sa se conecteze la un server pe adresa “localhost” portul 12345.
Clientul va citi de la server obiecte de tipul “Mesaj” pe care le va afisa la consola in forma urmatoare:
sender (dataMesaj) : mesaj.
De exemplu:
Mihaela (10.10.2018 12:34) : Salutare lume!
dupa care va trimite un singur mesaj la server cu un continut pe care doriti voi (hardcodat) si apoi va inchide conexiunea.

## Exercitiul 3

Construiti o interfata grafica pentru client, astfel incat sa avem un textfield cu numele senderului, un textfield cu mesajul de trimis, un buton de send si un JTextArea cu mesajele primite de la ceilalti.
Integrati aceasta interfata grafica cu clientul de la exercitiul 2 si faceti sa poata fi trimise mai multe mesaj.

## Exercitiul 4

Modificati serverul astfel incat sa suporte mai multi clienti conectati (folosind fire de executie).
Modificati lista interna de mesaje, astfel incat toti clientii sa poata sa adauge mesaje in aceeasi lista si atunci cand un client adauga un mesaj, mesajul sa fie apoi trimis catre toti clientii pentru ca sa apara in chat.