# Chapitre 6

## La communication Client-Serveur

Architecture client-serveur
Utilisation des sockets en Java

**1. INTRODUCTION 2**

**2. DEFINITIONS 4**

**2.1. LA COMMUNICATION CLIENT-SERVEUR 4**
**2.2. LE SOCKET 5**

**3. LES SOCKETS EN JAVA 6**

**3.1. EXEMPLE: CAS 1 (EXEMPLE 33 SUR LE SITE NFA 032) 6**
**3.2. EXEMPLE : CAS 2 (EXEMPLE 33 SUR LE SITE NFA 032) 8**
**3.3. EXEMPLE : AVEC UNE IHM (EXEMPLE 35 SUR LE SITE NFA 032) 9**
**3.4. EXEMPLE : CAS 3 (EXEMPLE 33 SUR LE SITE NFA 032) 9**
**3.5. EXEMPLE : CAS 3BIS (EXEMPLE 33 SUR LE SITE NFA 032) 11**
**3.6. EXEMPLE : CAS 4 (EXEMPLE 33 SUR LE SITE NFA 032) 11**

**4. LA COMMUNICATION DES OBJETS SUR UN SOCKET : EXEMPLE 34 DU SITE NFA 032 13**

**4.1. CAS 1 13**
**4.2. CAS 2 13**
**4.3. CAS 3 13**
**4.4. CAS 4 13**
**4.5. CAS5 14**
**4.6. CAS6 14**

**5. LA SECURISATION EN SALLE DE TP DU CNAM 14**

**6. LE DP OBSERVATEUR DISTANT** ERREUR ! SIGNET NON DEFINI.

**7. LE MVC DISTANT** ERREUR ! SIGNET NON DEFINI.

# 1. Introduction

JAVA est un langage de programmation qui assure des services permettant de couvrir les besoins des applications intranet/internet et serveur/client. Les propriétés de robustesse, d'encapsulation inhérantes aux mécanismes du langage ne fait que rendre ces services plus facilement accessibles par rapport aux autres langages.

Héritant du meilleur des ses prédécesseurs, JAVA est prêt à répondre aux exigences imposées par les nouvelles applications informatiques dont les caractéristiques sont :

- des postes clients banalisés par leurs interfaces d'accès (internet, intranet);
- la communication par réseau;
- la création de service distribué qui cohabite avec la plupart des systèmes informatiques existants;
- la création de nouvelles architectures d'applications avec la technologie Internet dont l'implémentation de modèle client-serveur et l'utilisation de bus logiciels de type CORBA, RMI, ...

Les architectures intranet peuvent se séparer en 2 grandes familles :

- les architectures deux niveaux;
- Les architectures trois ou multi-niveaux.

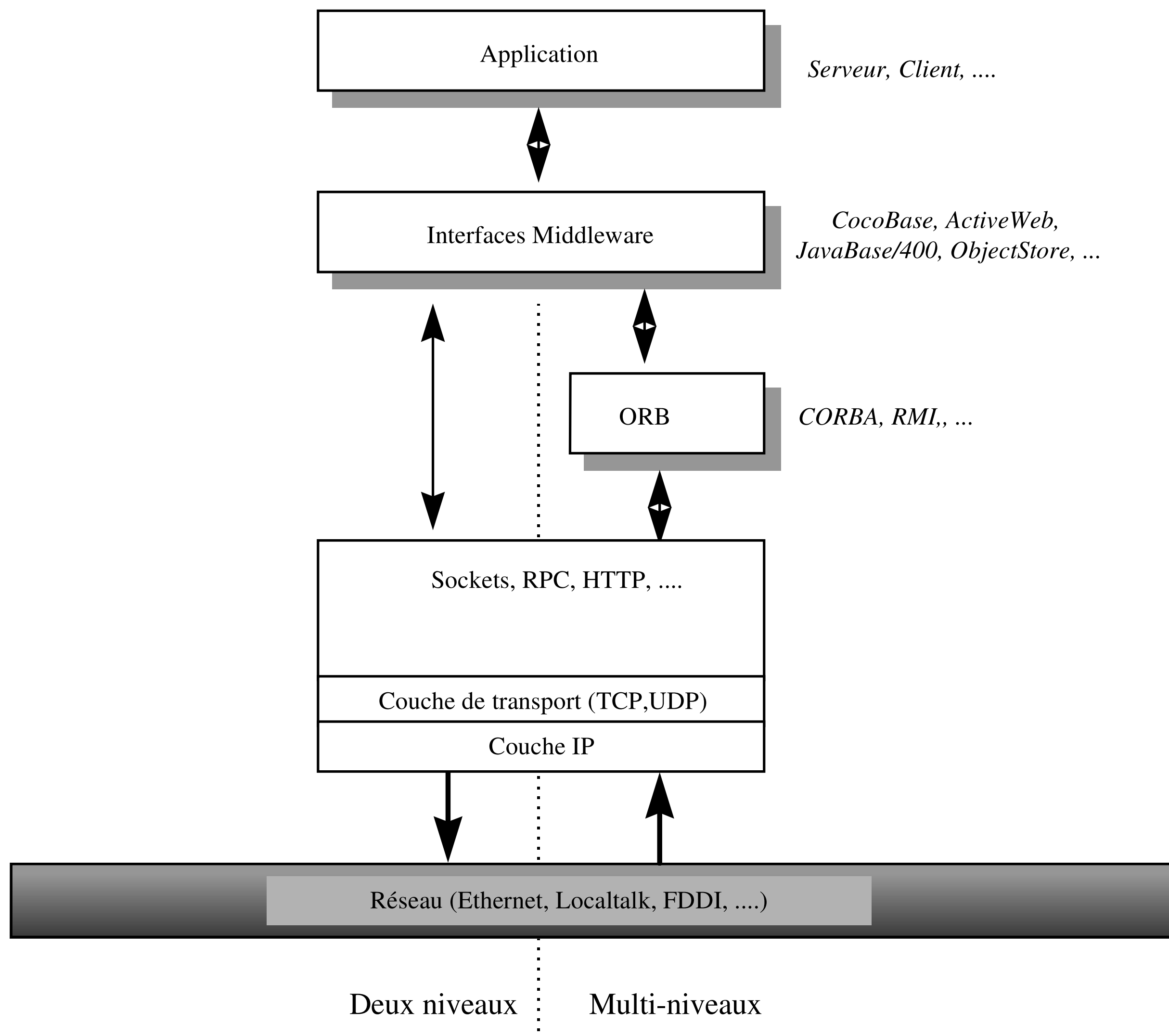
Elles reposent toutes deux sur la notion de **Middleware**.

Un "Middleware" (littéralement: "intermédiaire d'articles fabriqués") est un ensemble de composants logiciels assurant les interfaces de communication des données et l'appel éventuel aux traitements.

Un middleware s'appuie sur un empilement de couches logiciels plus ou moins sophistiquées assurant la communication physique des informations. Le niveau d'encapsulation et d'abstraction de cette pile de couches logiciels est déterminant dans la réalisation de l'ensemble afin de maîtriser notamment :

- le coût de développement
- la robustesse et l'évolution
- la facilité de mise en œuvre

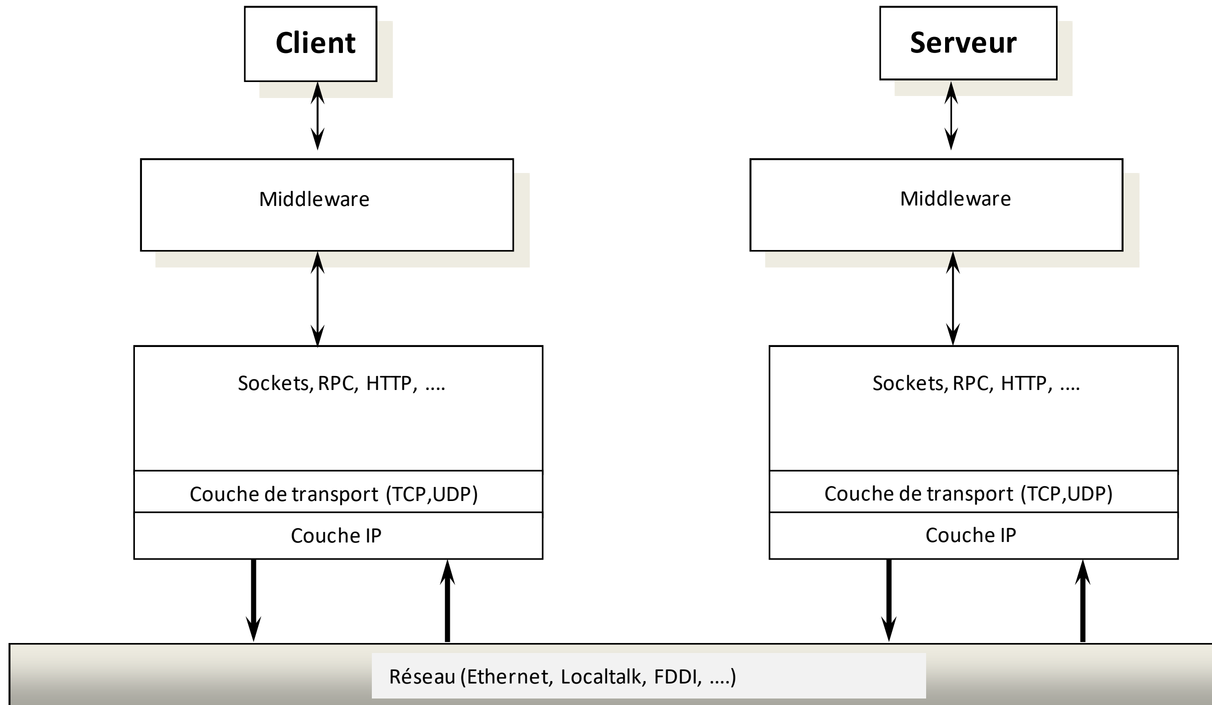
L'architecture d'un Middleware peut se schématiser de la manière suivante:




(ORB = Object Request Broker)

Un Middleware est une couche logicielle (ensemble de classes prédéfinies et d'eéxcutables) dont les traitements et les données (objets et méthodes de cet ensemble de classes) permettent de :

- définir un choix d'encodage et de décodage des données afin qu'elles soient transportables à travers le réseau : le format de transport des données . Comment les données vont-elles être écrites sur le socket.
- définir la logique de la communication de ces données (code de retour, retour des exceptions, communication synchrone ou asynchrone)
- Qui est serveur et qui est client ?
- Communication par push ou pull
- Identification et localisation des services (annuaires)
- … etc ….




Notre objectif dans cette formation n'est pas de faire un cours sur l'utilisation des Middleware existant ou des ORB, ou l'implémentation d'un Middleware, mais de montrer comment on utilise les sockets dans un langage de programmation car ces derniers sont utilisés dans l'implémentation d'un Middleware.

# 2. Définitions

## *2.1. La communication client-serveur*

Elle est la base de toute communication sur un réseau.

Cela va de soit, mais il est mieux de le préciser : la communication entre un client et un serveur se fait dans les deux sens.

Au début, il existe toujours une première entité qui est **en attente** d'une certaine communication : on l'appelle le **serveur**.

Puis un deuxième programme va **se connecter** (il lui envoie une donnée) au serveur : on l'appelle le **client**.

Scénario 1 (communication synchrone, mode déconnecté):
Le serveur est en attente.
Le client ouvre une connexion avec le serveur, lui envoie une donnée et se met en attente de la réponse.
Le serveur réalise son traitement et envoie la réponse au client.
Le serveur ferme la connexion et se remet en attente.

Scénario 2 (communication asynchrone, mode déconnecté):
Le serveur est en attente.

Le client ouvre une connexion avec le serveur, lui envoie une donnée et ne se met pas en attente d'une réponse.
Le serveur réalise son traitement.
Le serveur ferme la connexion et se remet en attente.

Scénario 3 (communication synchrone, mode connecté):
Le serveur est en attente.
Le client ouvre une connexion avec le serveur, lui envoie une donnée et se met en attente de la réponse
Le serveur réalise son traitement et envoie la réponse au client.
Le client envoie une autre donnée.
Le serveur réalise son traitement et envoie la réponse au client.
….
Le serveur ferme la connexion et se remet en attente.

## 2.2. Le Socket

Socket (mot anglais qui signifie prise) est un terme informatique qui peut avoir plusieurs significations suivant s'il est utilisé dans le cadre logiciel ou matériel.

Dans le contexte logiciel, le socket est un canal de communication permettant de faire communiquer deux logiciels distants et pour lequel un certain nombre de services sont utilisables. On parle d'établir une session TCP (Protocole de transport)

Dans le contexte matériel, d'une communication inter processus (IPC = Inter Process Communication) afin de permettre à divers processus de communiquer aussi bien sur une même machine qu'à travers un réseau TCP/IP.

La communication par socket est souvent comparée aux communications humaines. On distingue ainsi deux modes de communication :

- Le mode connecté (comparable à une communication téléphonique), utilisant le protocole TCP. Dans ce mode de communication, une connexion durable est établie entre les deux processus, de telle façon que l'adresse de destination n'est pas nécessaire à chaque envoi de données.
- Le mode non connecté (analogue à une communication par courrier), utilisant le protocole UDP. Ce mode nécessite l'adresse de destination à chaque envoi, et aucun accusé de réception n'est donné.

Dans beaucoup d'applications traditionnelles écrites notamment en C/C++, on retrouve l'utilisation des sockets pour implémenter la communication des informations entre le serveur et le client. Un des moyens les plus simples de mise en œuvre est l'utilisation de sockets.

Le socket est un "canal" de communication entre deux processus se trouvant ou non sur la même machine. Il est caractérisé par :

- l'addresse et
- le port de la machine vers laquelle on envoie un flot d'information

L'adresse est un moyen permettant d'identifier de manière unique la machine destinataire : c'est l'adresse IP.

Le port est un **numéro interne** à la machine lui permettant d'être en interrogation sur plusieurs ports différents (ex: un port pour le navigateur, un port pour la messagerie, …).

Au sens de la carte réseau, la notion de port est bien une notion logique, géré par le système d'exploitation. Un port est une sorte de porte ouverte par l'OS au monde extérieur.

Ainsi, une machine peut dialoguer en parallèle (si l'OS est multi-tâche) avec différentes autres machines.

Par contre, sur une même machine, un seul process à la fois ne peut être serveur sur un port.

Bien sur, sur une même machine, plusieurs process peuvent être à la fois client sur un port.

Des plages de port sont « réservés » (c'est une convention et non une obligation matérielle) de 0 à 1024. Exemples :

2, pour l'accès à un shell sécurisé Secure SHell, également utilisé pour l'échange de fichiers sécurisés SFTP
23, pour le port telnet
25, pour l'envoi d'un courrier électronique via un serveur dédié SMTP
53, pour la résolution de noms de domaine en adresses IP : DNS
67/68, pour DHCP et bootpc
80, pour la consultation d'un serveur HTTP par le biais d'un Navigateur web
110, pour la récupération de son courrier électronique via POP
123 pour la synchronisation de l'horloge : Network Time Protocol (NTP)
143, pour la récupération de son courrier électronique via IMAP
389, pour la connexion à un LDAP
443, pour les connexions HTTP utilisant une surcouche de sécurité de type SSL : HTTPS

Le principe du Firewall est d'autoriser et filtrer l'accès à ces ports.

# 3. Les sockets en JAVA

Les classes utilisés sont **Socket** et **ServerSocket** du package **java.net**.

La classe *ServerSocket* permet côté de serveur de se mettre en attente de la réception d'un premier flot de donnée sur un port donné. Chaque connexion se traduit par la création d'un *socket*.

La classe *Socket* permet de définir un objet **InputStream** et **OutputStream** afin de lire ou d'écrire des informations. Un socket se manipule donc comme un fichier.

## *3.1. Exemple: cas 1 (Exemple 33 sur le site NFA 032)*

Le serveur attend une seule requête du client. Le client envoie une seule requête.
Le client est en attente de lecture de la réponse.

**Le serveur :**

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

public class Serveur1
{
   public static void main(String args[]) throws Exception
   {
         String str;
         ServerSocket ssoc;
         ssoc = new ServerSocket(9999);
```

```
        System.out.println("En attente...");
        Socket soc = ssoc.accept();
        System.out.println("Socket accepte");

        InputStream is = soc.getInputStream();
        DataInputStream dis = new DataInputStream(is);

        System.out.println("Lecture du socket");
        str = dis.readUTF();
        System.out.println("RECU: "+str);

        soc.close();

    }
}
```

**Le client :**

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

public class Client1
{
   static public void main(String args[]) throws Exception
   {
        System.out.println("Creation du socket");
        Socket soc = new Socket("localhost",9999);

        OutputStream os=soc.getOutputStream();
        DataOutputStream dos=new DataOutputStream(os);

        DataInputStream in = new DataInputStream(System.in);
        System.out.print("> ");
        System.out.flush();
        String valeur= in.readLine();

        dos.writeUTF(valeur);

        soc.close();
   }
}
```

Le serveur est en attente de la part d'un client de la création d'un socket:
ssoc = new ServerSocket(9999); Le serveur attend sur le port 9999 à son adresse IP
Socket soc = ssoc.accept();
Le client crée le socket :
Socket soc = new Socket(InetAddress.getLocalHost(),9999);
Le premier paramètre est l'adresse IP du serveur. On peut utiliser :
"localhost", "192.168.0.1", InetAddress.getLocalHost() pour désigner l'IP du serveur du poste local, ou
l'adresse IP du serveur si le serveur n'est pas sur la même machine que le client.
L'instruction de lecture readUTF sur le socket est en attente d'écriture du client :
str = dis.readUTF();

## 3.2. Exemple : cas 2 (Exemple 33 sur le site NFA 032)

Le serveur attend de multiples requêtes du client. Le client envoie de multiples requêtes au serveur. Le client ne reçoit pas de retour du serveur.

**Le serveur :**

```
public class Serveur2
{
   public static void main(String args[]) throws Exception
   {
        String str;
        ServerSocket ssoc;
        ssoc = new ServerSocket(9999);

        System.out.println("En attente...");
        Socket soc = ssoc.accept();
        System.out.println("Socket accepte");

        InputStream is = soc.getInputStream();
        DataInputStream dis = new DataInputStream(is);
```

```
         while(true)
              {
                   System.out.println("Lecture du socket");
                   str = dis.readUTF();
                   System.out.println("RECU: "+str);
              }
   }
}
```

**Le client :**

```
public class Client2
{
   static public void main(String args[]) throws Exception
   {
         Socket soc = new Socket(InetAddress.getLocalHost(),9999);

         OutputStream os=soc.getOutputStream();
         DataOutputStream dos=new DataOutputStream(os);

        while(true)
              {
                   DataInputStream in = new DataInputStream(System.in);
                   System.out.print("> ");
                   System.out.flush();
                   String valeur= in.readLine();

                   dos.writeUTF(valeur);
              }
   }
}
```

Le serveur boucle sur la lecture du socket.
Le client boucle sur l'écriture du socket.

## 3.3. Exemple : avec une IHM (Exemple 35 sur le site NFA 032)

Le même cas de fonctionnement que le cas précédent mais avec une Ihm côté client et côté serveur afin d'afficher les informations saisies et reçues.

## 3.4. Exemple : cas 3 (Exemple 33 sur le site NFA 032)

Identique à l'exemple précédent mais le serveur répond au client comme quoi il a traité la requête du client.

On voit ici que la communication sur un socket se fait bien dans les deux sens.

**Le serveur :**

```
public class Serveur3
{
   public static void main(String args[]) throws Exception
   {
        String str;
        ServerSocket ssoc;
        ssoc = new ServerSocket(9999);

        System.out.println("En attente...");
        Socket soc = ssoc.accept();
        System.out.println("Socket accepte");

        InputStream is = soc.getInputStream();
        OutputStream os = soc.getOutputStream();
        DataInputStream dis = new DataInputStream(is);
        DataOutputStream dos = new DataOutputStream(os);

        while(true)
             {
                  System.out.println("Lecture du socket");
                  str = dis.readUTF();
                  System.out.println("RECU: "+str);
                  dos.writeUTF("RECU");
                  dos.writeUTF("RECU2"); // Pour faire un test
             }
   }
}
```

**Le client :**

```
public class Client3
{
   static public void main(String args[]) throws Exception
   {
        Socket soc = new Socket(InetAddress.getLocalHost(),9999);

        OutputStream os=soc.getOutputStream();
        InputStream is = soc.getInputStream();
        DataOutputStream dos=new DataOutputStream(os);
        DataInputStream dis = new DataInputStream(is);

       while(true)
             {
                  DataInputStream in = new DataInputStream(System.in);
                  System.out.print("> ");
                  System.out.flush();
                  String valeur= in.readLine();

                  dos.writeUTF(valeur);

                  String rep = dis.readUTF();
                  System.out.println("REPONSE: "+ rep);
             }
   }
}
```

On voit bien dans ce dernier exemple que la communication se fait bien dans les 2 sens.

## *3.5. Exemple : cas 3bis (Exemple 33 sur le site NFA 032)*

Dans le serveur, la ligne dos.writeUTF("RECU2"); (QU'IL FAUT DECOMMENTER ET RECOMPILER) permet de montrer que le socket est géré sous la forme d'un buffer (flot de données). Les données envoyées par le serveur s'accumulent dans le socket.

Si on veut que le client lise toutes les réponses du serveur :

```
public class Client3bis
{
   static public void main(String args[]) throws Exception
   {
        Socket soc = new Socket(InetAddress.getLocalHost(),9999);

        OutputStream os=soc.getOutputStream();
        InputStream is = soc.getInputStream();
        DataOutputStream dos=new DataOutputStream(os);
        DataInputStream dis = new DataInputStream(is);

       while(true)
             {
                  DataInputStream in = new DataInputStream(System.in);
                  System.out.print("> ");
                  System.out.flush();
                  String valeur= in.readLine();

                  dos.writeUTF(valeur);

                  String rep;
                  boolean lecture=true;
                  while(lecture)
                        {
                              rep = dis.readUTF();
                              System.out.println("REPONSE: "+ rep);
                              if(dis.available()==0)lecture=false;
                        }
             }
   }
}
```

La boucle while(lecture) permet de lire toutes les informations écrites sur le socket par le serveur.

Ceci met en évidence un problème de fond: le "protocole logique" que le client et le serveur doivent adopter en fonction des informations échangées.

Cette problématique peut être réduite en typant les informations échangées. Nous verrons que dans une architecture distribuée cette problématique tombe d'elle même.

## *3.6. Exemple : cas 4 (Exemple 33 sur le site NFA 032)*

Dans les deux exemples précédents on peut s'apercevoir que le serveur ne peut pas gérer plus de un client. Si on lance un deuxième client, il est en attente de l'instruction "accept" du serveur de socket qui n'arrivera jamais puisqu'en dehors de la boucle.

La solution est, sur le serveur, de mettre le "accept" dans la boucle. Le client demande alors à chaque requête un nouveau socket.

**<u>Le Serveur</u>** :

```
public class Serveur4
```

```
{
   public static void main(String args[]) throws Exception
   {
         String str;
         ServerSocket ssoc;
         ssoc = new ServerSocket(9999);

         while(true)
               {
                   System.out.println("En attente...");
                   Socket soc = ssoc.accept();
                   System.out.println("Socket accepte");

                   InputStream is = soc.getInputStream();
                   DataInputStream dis = new DataInputStream(is);

                    System.out.println("Lecture du socket");
                    str = dis.readUTF();
                    System.out.println("RECU: "+str);

                   soc.close();
               }
   }
}
```

**Le Client** :

```
public class Client4
{
   static public void main(String args[]) throws Exception
   {

        while(true)
               {

                  DataInputStream in = new DataInputStream(System.in);
                  System.out.print("> ");
                  System.out.flush();
                  String valeur= in.readLine();

                  Socket soc = new Socket(InetAddress.getLocalHost(),9999);

                   OutputStream os=soc.getOutputStream();
                   DataOutputStream dos=new DataOutputStream(os);

                   dos.writeUTF(valeur);

                   soc.close();
               }
   }
}
```

# 4. La communication des objets sur un socket : Exemple 34 du site NFA 032

Typer une information consiste à échanger des données qui correspondent à des classes d'objet. On réalise ce mode de transfert avec le **principe de sérialisation**.

Cet exemple montre des exemples d'utilisation de socket dans le cadre d'une communication d'objet en utilisant le principe de sérialisation.

~~Les cas 1 et 2 introduisent le sujet. Ces deux cas n'utilisent pas le principe de sérialisation.~~

Les autres cas utilisent le principe de sérialisation.

Tous les cas de cet exemple se compilent et s'exécutent de la manière suivante :
Compilations :
javac Client.java
javac Serveur.java
Exécutions:
java Serveur
java Client (dans une autre fenêtre)

## *4.1. Cas 1*

On veut envoyer un objet java de classe Individu (nom, prénom, age).
Le client écrit chaque champ de l'individu et le serveur les lit champ à champ.

## *4.2. Cas 2*

Utilisation du principe de sérialisation.
La classe Individu implements l'interface Serializable.
Le flot de lecture et d'écriture n'est plus un DataOutputStream (resp DataInputStream) mais un ObjectOutputSream (resp. ObjectInputStream).
L'encodage de l'objet dans le socket devient transparent mais l'objet écrit et lu est de la classe Object.

## *4.3. Cas 3*

Le serveur peut recevoir des objets de classes différentes. Exemple: Individu ou Complexe.
Si il est nécessaire de créer le véritable objet, il est indispensable d'utiliser l'opérateur **instanceof** afin de tester la classe d'appartenance.
Dans le sous-cas Serveur2, il est inutile d'utiliser instanceof car la méthode appelée (toString) est générique pour les deux classes.

## *4.4. Cas 4*

Le client et le serveur s'échangent les objets Java en utilisant le XML.
Dans ce cas de sérialisation, il faut ouvrir un nouveau socket entre chaque requête.

La sérialisation utilisée est celle native de Java : java.beans

Le package java.beans permet le développement de "beans".
Un beans est ....
Une propriété essentielle d'un beans est de pouvoir être échanger entre un serveur et un client. Le serveur contient des beans devant être déployés sur des postes clients. Ce serveur est la plupart du temps un serveur HTTP et les clients des environnements d'exécution Java (Applet, JavaWebStart, ...). Pour cela, le beans doit transiter par un flux de données (socket), le beans doit donc être sérialisé puis dessérialisé. De plus, les

environnements d'exécution du client et du serveur peuvent être de version java différente. Il faut donc rendre robuste la sérialisation en n'utilisant pas le principe binaire de sérialisation des objets Java mais un principe de sérialisation dont le format soit textuel (universel) et dont la syntaxe soit la plus ouverte possible. Le format tout trouvé est le XML.

C'est ainsi que le package java.beans possède les classes XMLDecoder et XMLEncoder permettant de sérialiser et dessérialiser n'importe quel beans.

En Java, un beans est une classe Java :

- qui possède au moins un constructeur sans paramètre
- dont certains attributs privés possèdent les méthodes d'accès et de modification (get et set)

Seuls les attributs possédant ces accesseurs et ces assesseurs sont sérialisés au format XML.

### *4.5. Cas5*

Un cas qui fonctionne en encodant toutes les données en chaîne et en utilisant les encodeurs et les décodeurs XML avec la technologie Beans.

### *4.6. Cas6*

Idem cas précédent mais en utilisant un streamer DOM.

# 5. La sécurisation en salle de TP du CNAM

Voir l'exemple **Exemple48_SocketSecurity** sur le site du cours NFA 032.

Voir l'exemple **Exemple47_TalkUs** sur le site du cours NFA 032.

# 6. Serveur de socket et thread

3 points sont à aborder :

**1**/ Dans le cas où une application exécute plusieurs serveurs de socket, il est indispensable que les serveurs de socket s'exécutent dans un thread car pour chacun des serveurs de sockets, on crée une boucle « infini » dans laquelle le serveur de socket se met en attente d'une connexion, puis exécute des traitements une fois une connexion a été établie.

**2**/ Dans le cas où on veut qu'un client d'un serveur de socket ne soit pas bloqué par le traitement des autres clients, il est indispensable que le traitement de chacun des clients se fasse dans un thread.

Cela est nécessaire quand les traitements de chacun des clients est long et aussi quand le nombre de client est important.

NB : Nous ne sommes pas dans ce cas de figure pour notre projet.

**3**/ Dans le cas où plusieurs threads utilisent en parallèle le même périmètre d'applicatif, il est indispensable que les objets utilisés dans ce périmètre soient multi-threadés ou que toutes les méthodes (des mêmes objets) appelées dans chacun des threads soit synchronisées (Synchronized).

Explication et schémas en cours.