
Formation Java avancé

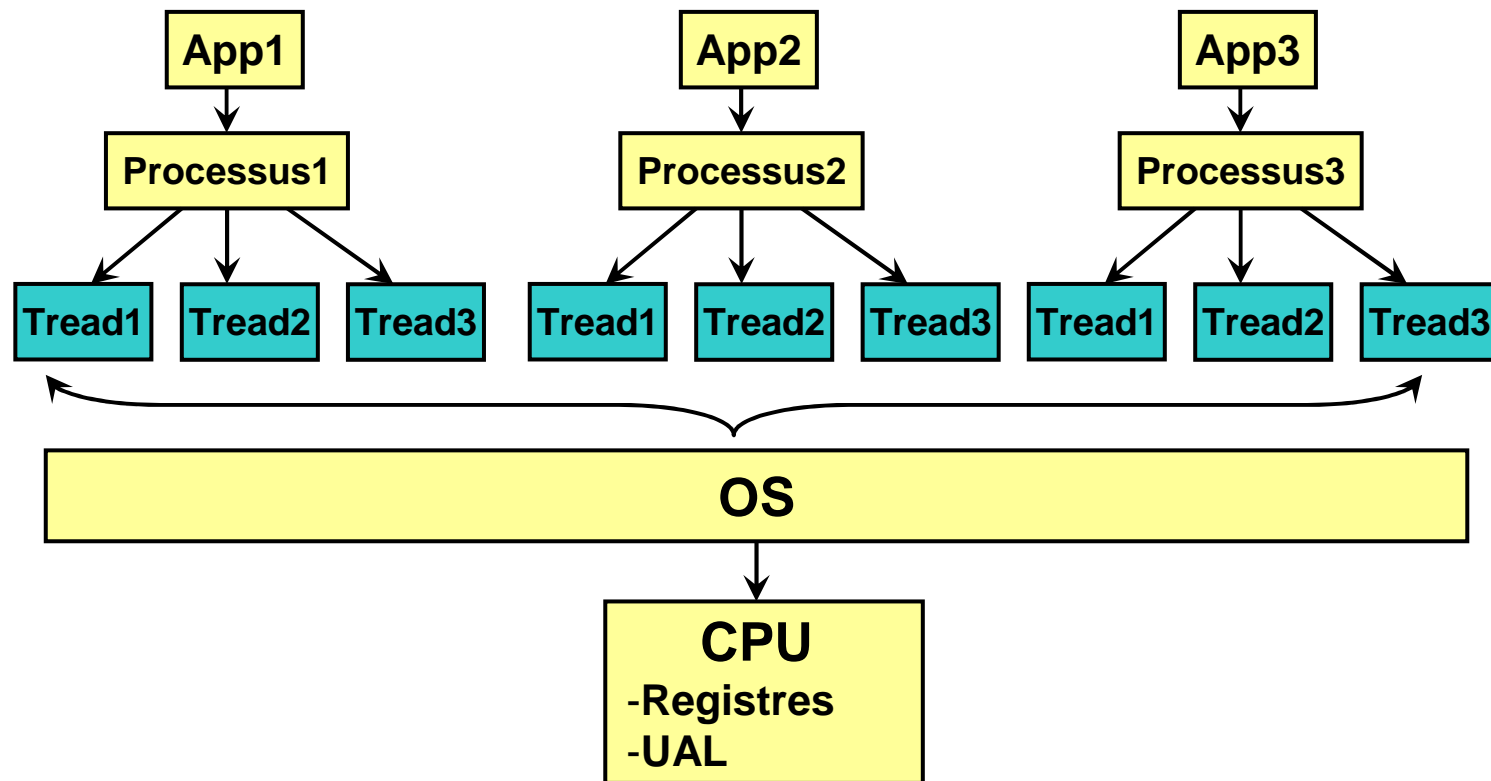
Chebihi Fayçal

F.chebihi @gmail.com

Les Threads en Java

Qu'est ce qu'un thread

- Java est un langage multi-threads
- Il permet d'exécuter plusieurs blocs d'instructions à la fois.
- Dans ce cas, chaque bloc est appelé Thread (tâche ou fil)



Création et démarrage d'un thread

En Java, il existe, deux techniques pour créer un thread:

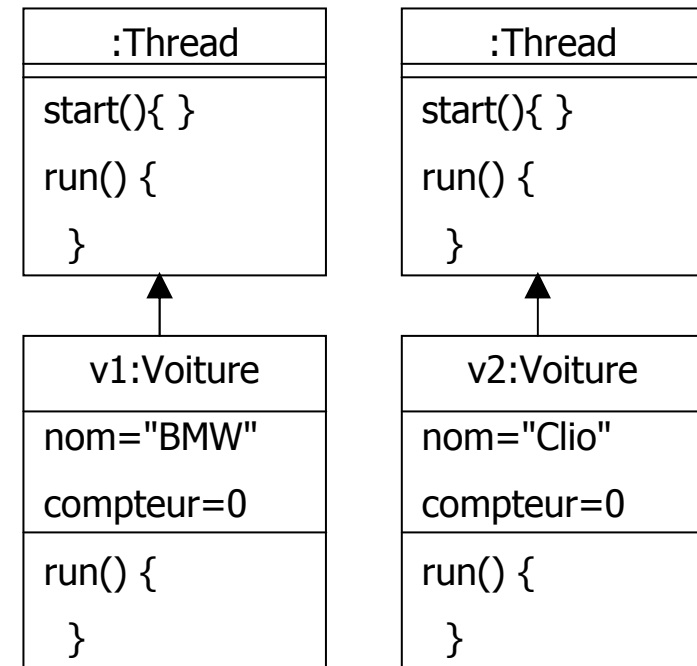
- Soit vous dérivez votre thread de la classe `java.lang.Thread`,
- soit vous implémentez l'interface `java.lang.Runnable`.

Créer un thread en dérivant de la classe java.lang.Thread.

```
public class ThreadClass extends Thread {
    // Les attributs de la classe
    // Les constructeurs de la classe
    // Méthode de la classe
    @Override
    public void run() {
        // Code exécuté par le thread
    }
    public static void main(String[] args) {
        ThreadClass t1=new ThreadClass();
        ThreadClass t2=new ThreadClass();
        t1.start();t2.start();
    }
}
```

Exemple de classe héritant de Thread

```
public class Voiture extends Thread {  
    private String nom;  
    private int compteur;  
    public Voiture(String nom) {  
        this.nom=nom;  
    }  
    public void run(){  
        try{  
for(int i=0;i<10;i++){  
            ++compteur;  
            System.out.println("Voiture:"+nom+"  
I="+i+"Compteur="+compteur);  
            Thread.sleep(1000);  
        }}catch(InterruptedException e){  
            e.printStackTrace(); }  
        }  
    public static void main(String[] args){  
        Voiture v1=new Voiture("BMW");  
        Voiture v2=new Voiture("Clio");  
        v1.start();  
        v2.start();  
    }  
}
```



Exécution :

Voiture BMW I=0 Compteur=1
Voiture Clio I=0 Compteur=1
Voiture BMW I=1 Compteur=2
Voiture Clio I=1 Compteur=2
Voiture BMW I=2 Compteur=3
Voiture Clio I=2 Compteur=3

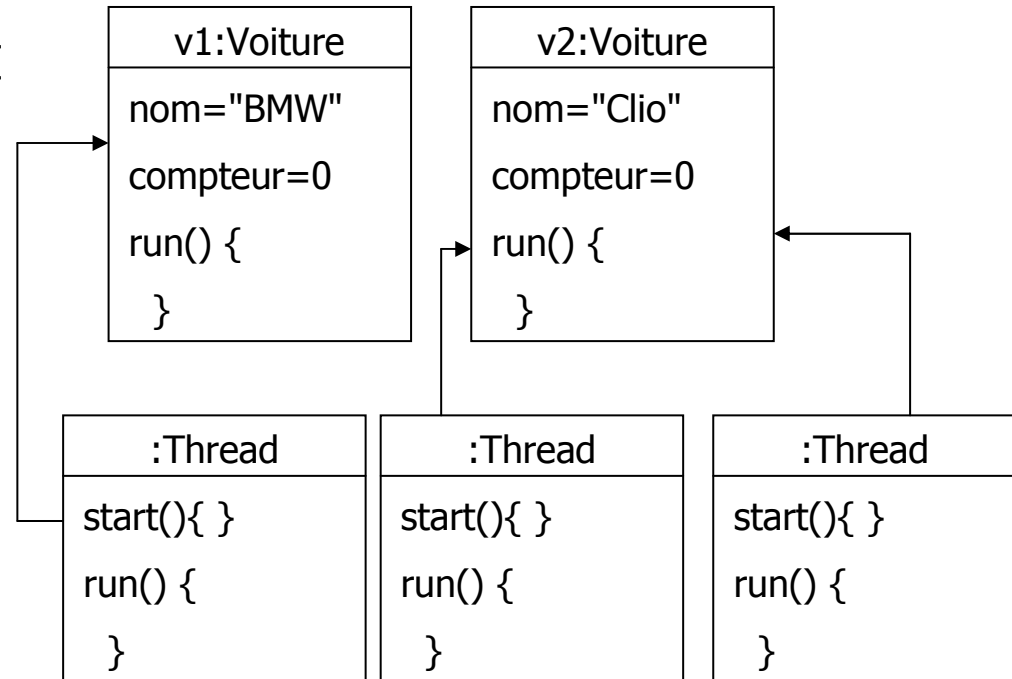
.....
Voiture BMW I=9 Compteur=10
Voiture Clio I=9 Compteur=10

Créer un thread en implémentant l'interface java.lang.Runnable.

```
public class ThreadClass implements Runnable {
    // Les attributs de la classe
    // Les constructeurs de la classe
    // Méthode de la classe
    @Override
    public void run() {
        // Code exécuté par le thread
    }
    public static void main(String[] args) {
        ThreadClass t1=new ThreadClass();
        ThreadClass t2=new ThreadClass();
        new Thread(t1).start();new Thread(t2).start();
    }
}
```

Exemple de classe implémentant Runnable

```
public class Voiture implements Runnable {
    private String nom;
    private int compteur;
    public Voiture(String nom) {
        this.nom=nom;
    }
    public void run(){
        try{
            for(int i=0;i<10;i++){
                ++compteur;
                System.out.println("Voiture:"+nom+"
I="+i+"Compteur="+compteur);
                Thread.sleep(1000);
            }catch(InterruptedException e){
                e.printStackTrace(); }
        }
    }
    public static void main(String[]largs){
        Voiture v1=new Voiture("BMW");
        Voiture v2=new Voiture("Clio");
        new Thread(v1).start();
        new Thread(v2).start();
        new Thread(v2).start();
    }
}
```



Exécution :

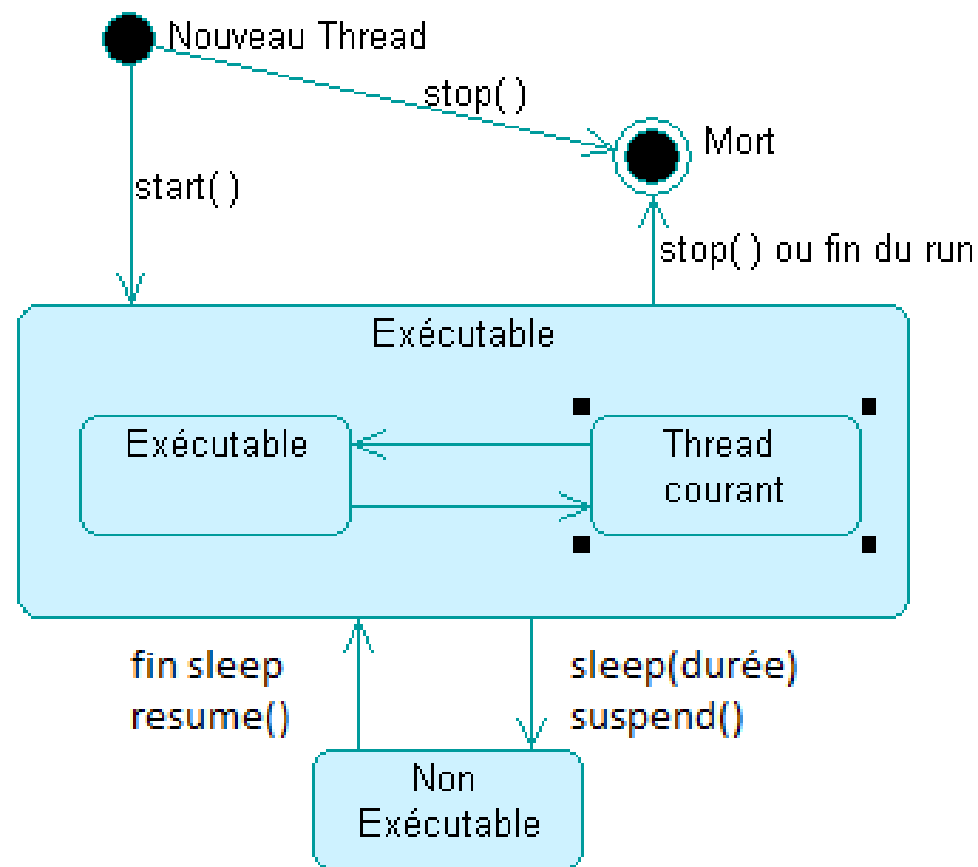
Voiture BMW I=0 Compteur=1
Voiture Clio I=0 Compteur=1
Voiture Clio I=0 Compteur=2
Voiture BMW I=1 Compteur=2
Voiture Clio I=1 Compteur=3
Voiture Clio I=1 Compteur=4
.....;

Quelle technique choisir ?

Méthode	<i>Avantages</i>	<i>Inconvénients</i>
extends Thread	Chaque thread a ses données qui lui sont propres.	On ne peut plus hériter d'une autre classe.
implements Runnable	L'héritage reste possible. En effet, on peut implémenter autant d'interfaces que l'on souhaite.	Les attributs de votre classe sont partagés pour tous les threads qui y sont basés. Dans certains cas, il peut s'avérer que cela soit un atout.

Gestion des threads

■ *Cycle de vie d'un thread*



Démarrage, suspension, reprise et arrêt d'un thread.

- **public void start()** : cette méthode permet de démarrer un thread. En effet, si vous invoquez la méthode *run* (au lieu de *start*) le code s'exécute bien, mais aucun nouveau thread n'est lancé dans le système. Inversement, la méthode *start*, lance un nouveau thread dans le système dont le code à exécuter démarre par le *run*.
- **public void suspend()** : cette méthode permet d'arrêter temporairement un thread en cours d'exécution.
- **public void resume()** : celle-ci permet de relancer l'exécution d'un thread, au préalable mis en pause via *suspend*. Attention, le thread ne reprend pas au début du *run*, mais continue bien là où il s'était arrêté.
- **public void stop()** : cette dernière méthode permet de stopper, de manière définitive, l'exécution du thread. Une autre solution pour stopper un thread consiste à simplement sortir de la méthode *run*.

Gestion de la priorité d'un thread.

- Vous pouvez, en Java, jouer sur la priorité de vos threads.
- Sur une durée déterminée, un thread ayant une priorité plus haute recevra plus fréquemment le processeur qu'un autre thread. Il exécutera donc, globalement, plus de code.
- La priorité d'un thread va pouvoir varier entre 0 et 10. Mais attention, il n'est en aucun cas garanti que le système hôte saura gérer autant de niveaux de priorités.
- Des constantes existent et permettent d'accéder à certains niveaux de priorités : `MIN_PRIORITY` (0) - `NORM_PRIORITY` (5) - `MAX_PRIORITY` (10).
- Pour spécifier la priorité d'un thread, il faut faire appel à la méthode `setPriority(int p)` de la classe `Thread`
- Exemple :
 - ❑ `Thread t=new Thread();`
 - ❑ `t.setPriority(Thread.MAX_PRIORITY);`

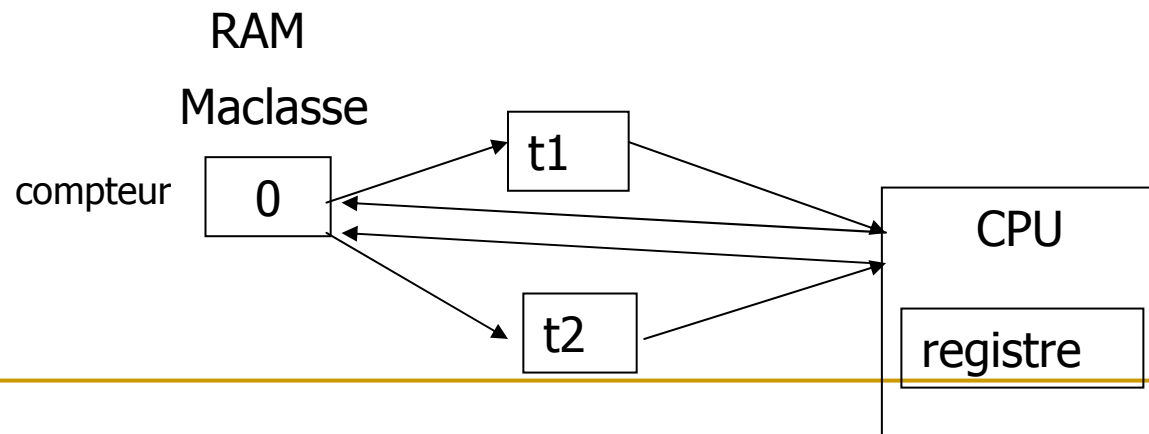
Synchronisation de threads et accès aux ressources partagées.

- Lorsque que vous lancez une JVM (Machine Virtuelle Java), vous lancez un processus.
- Ce processus possède plusieurs threads et chacun d'entre eux partage le même espace mémoire.
- En effet, l'espace mémoire est propre au processus et non à un thread.
- Cette caractéristique est à la fois un atout et une contrainte.
- En effet, partager des données pour plusieurs threads est relativement simple.
- Par contre les choses peuvent se compliquer sérieusement si la ressource (les données) partagée est accédée en modification
- il faut synchroniser les accès concurrents. Nous sommes certainement face à l'un des problèmes informatiques les plus délicats à résoudre.

Synchronisation de threads et accès aux ressources partagées.

- Soit l'exemple suivant:

```
public class MaClasse extends Thread{  
    private static int compteur;  
    public void run(){  
        compteur=compteur+1;  
        System.out.println(compteur);  
    }  
    public static void main(String[] Args){  
        MaClasse t1=new MaClasse();  
        MaClasse t2=new MaClasse();  
        t1.start(); t2.start();  
    }  
}
```



Synchronisation de threads et accès aux ressources partagées.

- Imaginez qu'un premier thread évalue l'expression *MaClasse.Compteur + 1* mais que le système lui hôte le cpu, juste avant l'affectation, au profit d'un second thread.
- Ce dernier se doit lui aussi d'évaluer la même expression. Le système redonne la main au premier thread qui finalise l'instruction en effectuant l'affectation, puis le second en fait de même.
- Au final de ce scénario, l'entier aura été incrémenté que d'une seule et unique unité.
- De tels scénarios peuvent amener à des comportements d'applications chaotiques.
- Il est donc vital d'avoir à notre disposition des mécanismes de synchronisation.

Notions de verrous

- L'environnement Java offre un premier mécanisme de synchronisation : les verrous (locks en anglais).
- Chaque objet Java possède un verrou et seul un thread à la fois peut verrouiller un objet.
- Si d'autres threads cherchent à verrouiller le même objet, ils seront endormis jusqu'à que l'objet soit déverrouillé.
- Cela permet de mettre en place ce que l'on appelle plus communément une section critique.
- Pour verrouiller un objet par un thread, il faut utiliser le mot clé **synchronized**.
- En fait, il y a deux façons de définir une section critique. Soit on synchronise un ensemble d'instructions sur un objet, soit on synchronise directement l'exécution d'une méthode pour une classe donnée.
- Dans le premier cas, on utilise l'instruction *synchronized* :

```
synchronized(object) {  
    //Instructions de manipulation d'une ressource partagée.  
}
```
- Dans le second cas, on utilise le qualificateur *synchronized* sur la méthode considérée:

```
public synchronized void meth(int param) {  
// Le code de la méthode synchronisée.}
```


Exemple : Synchroniser les threads au moment de l'incrément de la variable compteur

```
public class Voiture implements Runnable {
    private String nom;
    private int compteur;
    public Voiture(String nom) {
        this.nom=nom;
    }
    public void run(){
        try{
            for(int i=0;i<10;i++){
                synchronized (this) {
                    ++compteur;
                }
            }
            System.out.println("Voiture:"+nom+" I="+i+"Compteur="+compteur);
            Thread.sleep(1000);
        } catch (InterruptedException e){
            e.printStackTrace();
        }
    }
    public static void main(String[] args){
        Voiture v1=new Voiture("BMW"); Voiture v2=new Voiture("Clio");
        new Thread(v1).start();new Thread(v2).start();
        new Thread(v2).start();
    }
}
```

Exemple : Synchroniser les threads avant l'accès à la méthode run

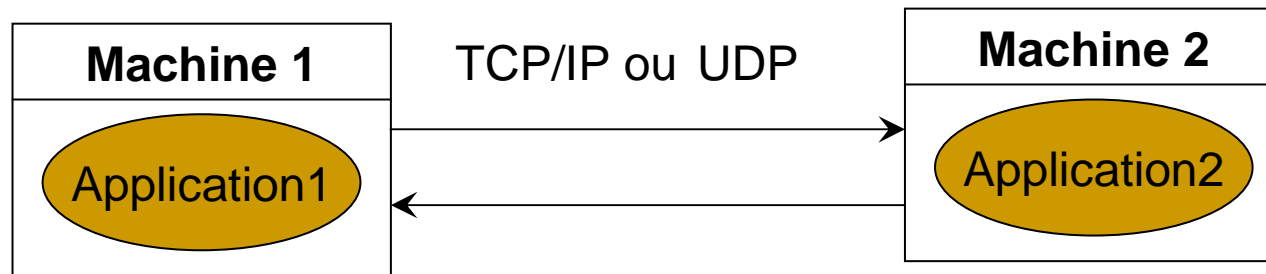
```
public class Voiture implements Runnable {
    private String nom;
    private int compteur;
    public Voiture(String nom) {
        this.nom=nom;
    }

    public synchronized void run(){
        try{
            for(int i=0;i<10;i++){
                ++compteur;
                System.out.println("Voiture:"+nom+" I="+i+"Compteur="+compteur);
                Thread.sleep(1000);
            }catch(InterruptedException e){
                e.printStackTrace(); }
        }

        public static void main(String[] args){
            Voiture v1=new Voiture("BMW"); Voiture v2=new Voiture("Clio");
            new Thread(v1).start();new Thread(v2).start();
            new Thread(v2).start();
        }
    }
}
```

Sockets dans java

Applications distribuées



Technologies d'accès :

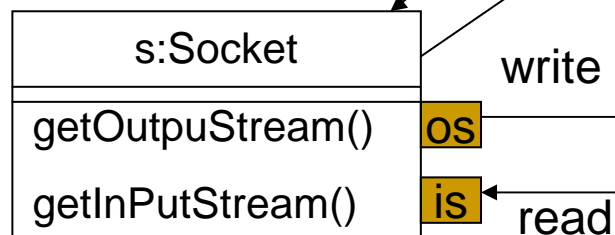
- Sockets ou DataGram
- RMI (JAVA)
- CORBA (Multi Langages)
- EJB (J2EE)
- Web Services (HTTP+XML)

Principe de base

Création d'un client :

```
Socket s=new Socket("192.168.1.23",1234)
```

```
InputStream is=s.getInputStream();  
OutputStream os=s.getOutputStream();  
os.write(23);  
int rep=is.read();  
System.out.println(rep);
```



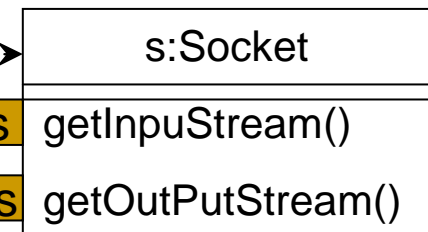
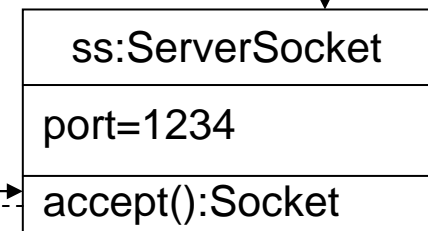
Création d'un serveur:

```
ServerSocket ss=new ServerSocket(1234);
```

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

```
InputStream is=s.getInputStream();  
OutputStream os=s.getOutputStream();  
int nb=is.read();  
int rep=nb*2;  
os.write(rep);
```

Connexion



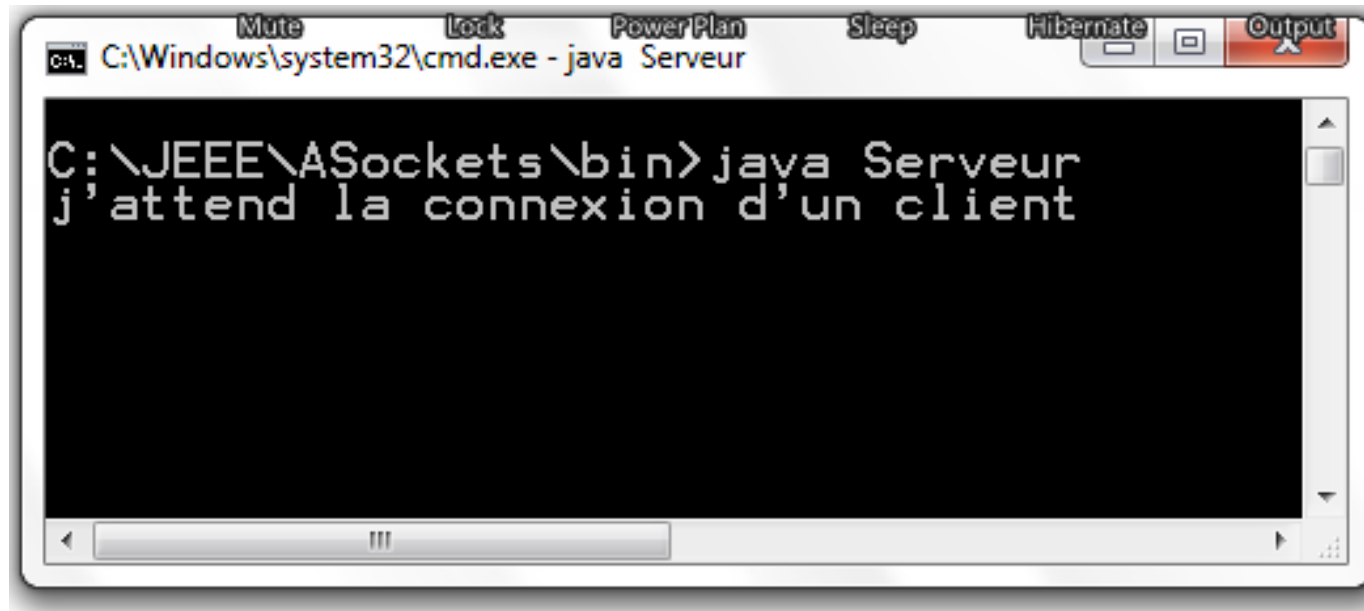
Exemple d'un serveur simple

```
import java.io.*;import java.net.*;
public class Serveur {
public static void main(String[] args) {
    try {
ServerSocket ss=new ServerSocket(1234);
System.out.println("j'attend la connexion d'un client");
Socket clientSocket=ss.accept();
System.out.println("Nouveau client connecté");
System.out.println("Génération de objet InptStream et
    OutputStream de la socket");
InputStream is=clientSocket.getInputStream();
OutputStream os=clientSocket.getOutputStream();
System.out.println("J'attend un nombre (1 octet)!");
int nb=is.read();
System.out.println("J'envoie la réponse");
os.write(nb*5);
System.out.println("Déconnexion du client");
clientSocket.close();
    } catch (IOException e) {
e.printStackTrace();
    }
}
}
```

Exemple d'un client simple

```
import java.io.*;import java.net.*;import java.util.Scanner;
public class Client {
    public static void main(String[] args) {
    try {
        System.out.println("Créer une connexion au serveur:");
        Socket clientSocket=new Socket("localhost", 123);
        System.out.println("Génération de objet InptStream et OutputStream
de la socket");
        InputStream is=clientSocket.getInputStream();
        OutputStream os=clientSocket.getOutputStream();
        System.out.print("Lire un nombre au clavier NB=");
        Scanner clavier=new Scanner(System.in);
        int nb=clavier.nextInt();
        System.out.println("Envoyer le nombre "+nb+" au serveur");
        os.write(nb);
        System.out.println("Attendre la réponse du serveur:");
        int rep=is.read();
        System.out.println("La réponse est :"+rep);
    } catch (Exception e) {
        e.printStackTrace();
    }
    }
}
```

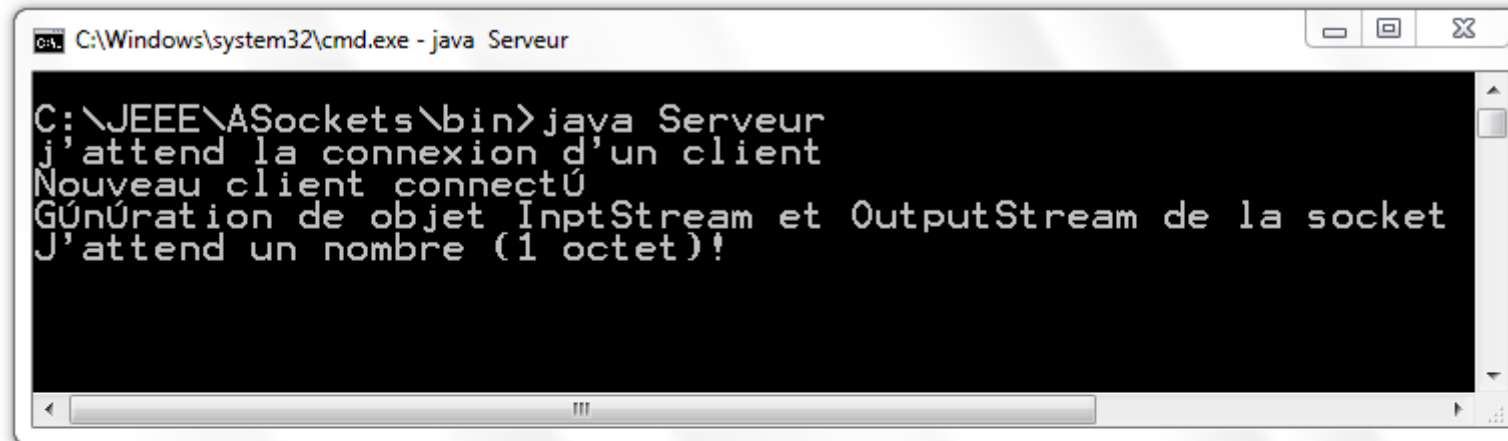
Lancement du serveur sur ligne de commandes



The image shows a Windows command prompt window. The title bar includes standard Windows window controls (minimize, maximize, close) and a status bar with 'Mute', 'Lock', 'Power Plan', 'Sleep', 'Hibernate', and 'Output' buttons. The address bar shows the command prompt is running 'C:\Windows\system32\cmd.exe - java Serveur'. The main text area displays the command 'C:\JEEE\ASockets\bin>java Serveur' and its output 'j'attend la connexion d'un client'. The window has a scroll bar on the right and a command history bar at the bottom.

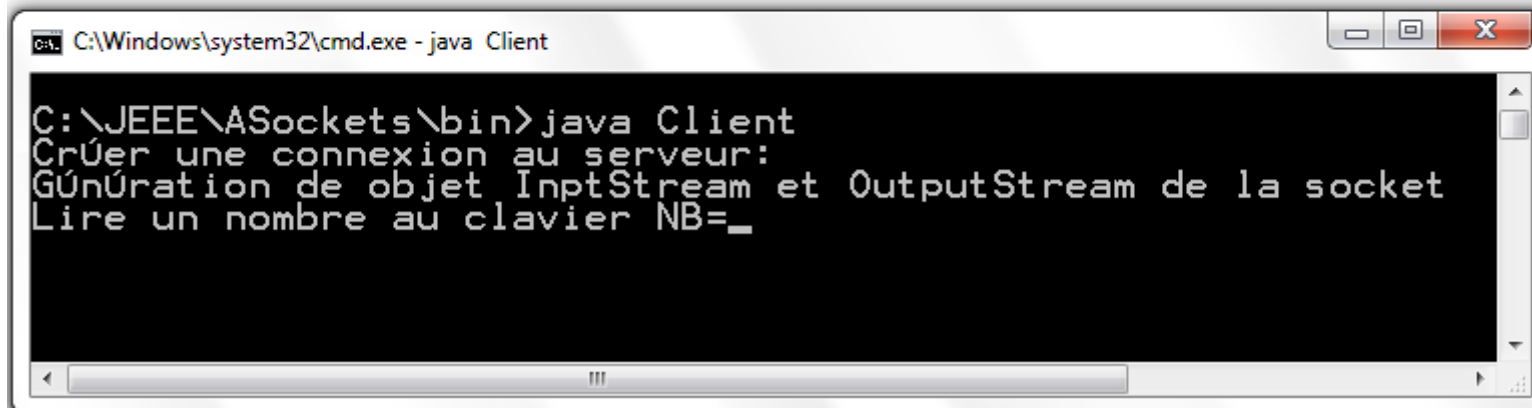
```
C:\Windows\system32\cmd.exe - java Serveur  
  
C:\JEEE\ASockets\bin>java Serveur  
j'attend la connexion d'un client
```


Lancement du client sur ligne de commandes



```
C:\Windows\system32\cmd.exe - java Serveur


C:\JEEE\ASockets\bin>java Serveur
j'attends la connexion d'un client
Nouveau client connecté
Génération de objet InptStream et OutputStream de la socket
J'attends un nombre (1 octet)!
```



```
C:\Windows\system32\cmd.exe - java Client

C:\JEEE\ASockets\bin>java Client
Créer une connexion au serveur:
Génération de objet InptStream et OutputStream de la socket
Lire un nombre au clavier NB=_
```

Saisir un nombre par le client et l'envoyer au serveur



```
C:\Windows\system32\cmd.exe

C:\JEEE\ASockets\bin>java Serveur
j'attend la connexion d'un client
Nouveau client connecté
Génération de objet InptStream et OutputStream de la socket
J'attends un nombre (1 octet)!
J'envoie la réponse
Déconnexion du client

C:\JEEE\ASockets\bin>
```



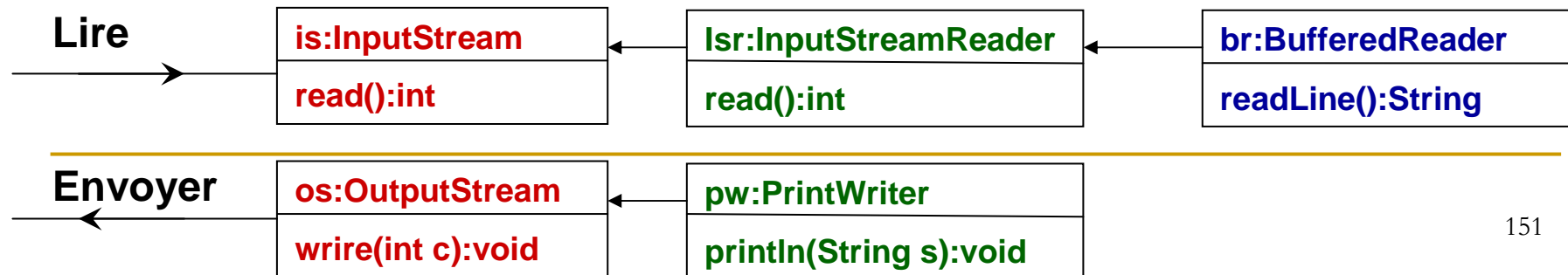
```
C:\Windows\system32\cmd.exe

C:\JEEE\ASockets\bin>java Client
Créer une connexion au serveur:
Génération de objet InptStream et OutputStream de la socket
Lire un nombre au clavier NB=12
Envoyer le nombre 12 au serveur
Attendre la réponse du serveur:
La réponse est :60

C:\JEEE\ASockets\bin>
```

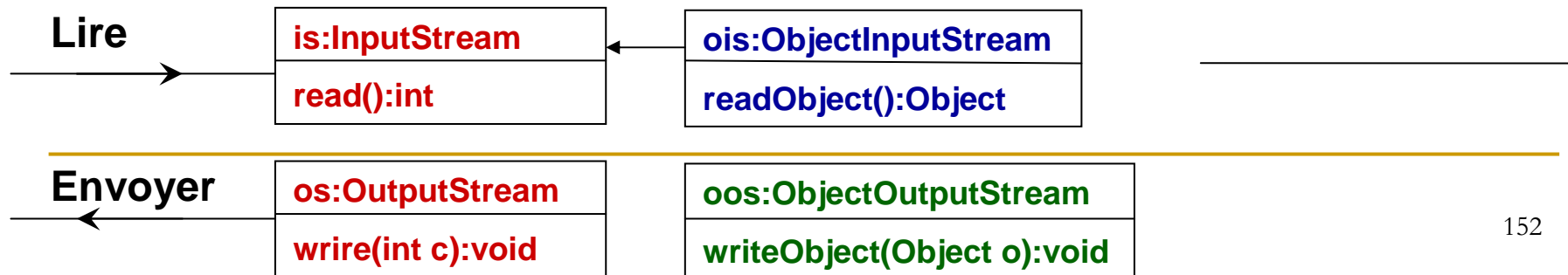
Recevoir et envoyer des chaînes de caractères

- Création de l'objet ServerSocket
 - `ServerSocket ss=new ServerSocket(2345);`
- Attendre une connexion d'un client
 - `Socket sock=ss.accept();`
- Pour lire une chaîne de caractère envoyée par le client :
 - `InputStream is=sock.getInputStream();`
 - `InputStreamReader isr=new InputStreamReader(is);`
 - `BufferedReader br=new BufferedReader(isr);`
 - `String s=br.readLine();`
- Pour envoyer une chaîne de caractères au client
 - `OutputStream os=sock.getOutputStream();`
 - `PrintWriter pw=new PrintWriter(os,true);`
 - `pw.println("Chaîne de caractères");`



R/E des objets (Sérialisation et désérialisation)

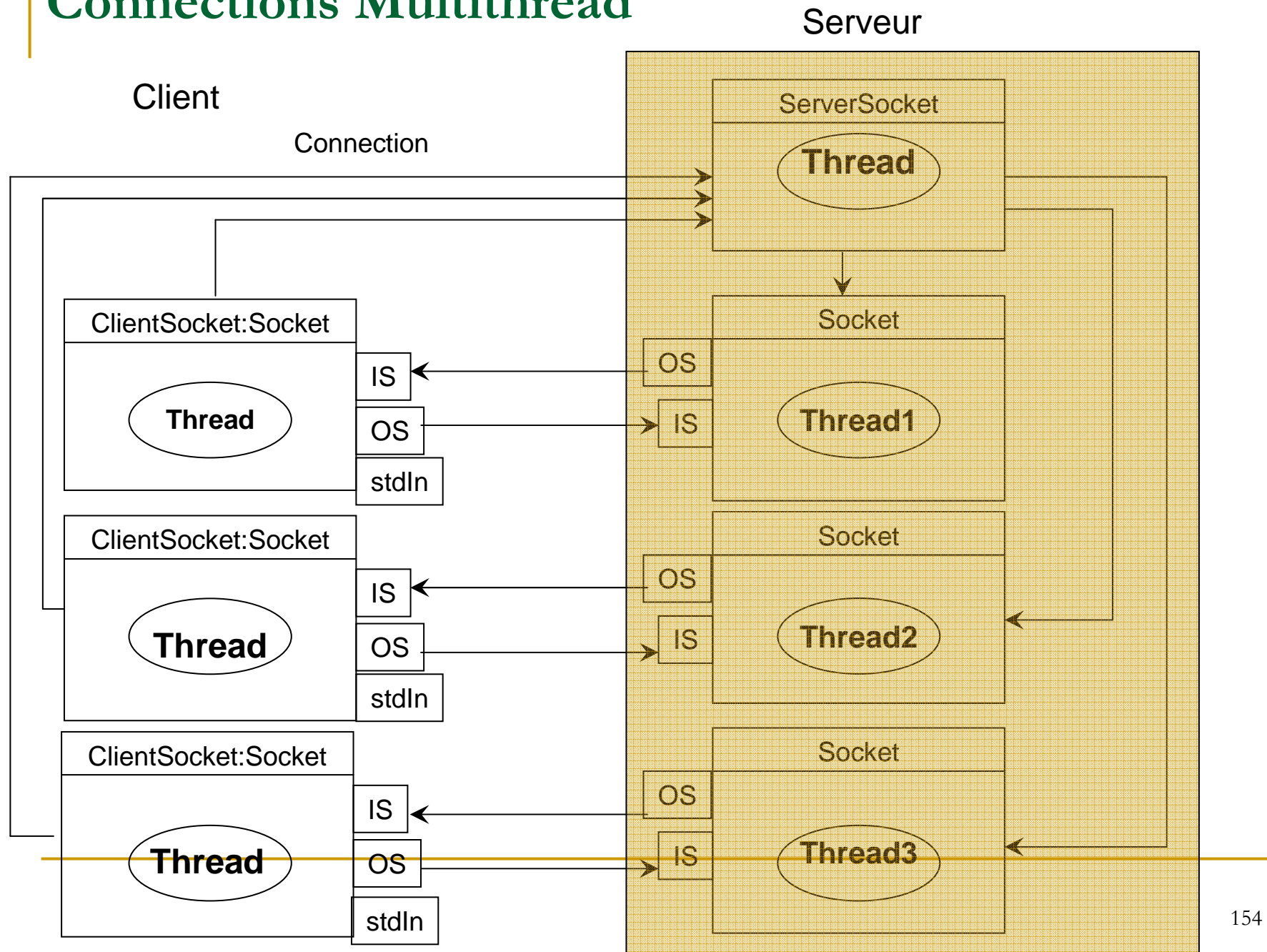
- Une classe Sérializable:
 - ❑ public class Voiture implements Serializable{
 - ❑ String mat;int carburant;
 - ❑ public Voiture(String m, int c) { mat=m; carburant=c;}
 - ❑ }
- Pour sérialiser un objet (envoyer un objet vers le client)
 - ❑ `OutputStream os=sock.getOutputStream();`
 - ❑ `ObjectOutputStream oos=new ObjectOutputStream(os);`
 - ❑ `Voiture v1=new Voiture("v212",50);`
 - ❑ `oos.writeObject(v1);`
- Pour lire un objet envoyé par le client (désérialisation)
 - ❑ `InputStream is=sock.getInputStream();`
 - ❑ `ObjectInputStream ois=new ObjectInputStream(is);`
 - ❑ `Voiture v=(Voiture) ois.readObject();`



Serveur Multi-Threads

- Pour qu'un serveur puisse communiquer avec plusieurs client en même temps, il faut que:
 - ❑ Le serveur puisse attendre une connexion à tout moment.
 - ❑ Pour chaque connexion, il faut créer un nouveau thread associé à la socket du client connecté, puis attendre à nouveau une nouvelle connexion
 - ❑ Le thread créé doit s'occuper des opérations d'entrées-sorties (read/write) pour communiquer avec le client indépendamment des autres activités du serveur.
-

Connections Multithread



Implémentation d'un serveur multi-thread

- Le serveur se compose au minimum par deux classes:

- `ServeurMultiThread.java` : serveur principal

```
import java.io.*; import java.net.*;
public class ServeurMultiThread extends Thread {
    int nbClients=0; // pour compter le nombre de clients connectés
    public void run(){
        try {
            System.out.println("J'attend une connexion");
            ServerSocket ss=new ServerSocket(4321);
            while(true){
                Socket s=ss.accept();
                ++nbClients;
                new ServiceClient(s,nbClients).start();
            }
        } catch (IOException e) {e.printStackTrace();}
    } // Fin de la méthode run
    public static void main(String[] args) {
        new ServeurMultiThread().start();
    }
}
```

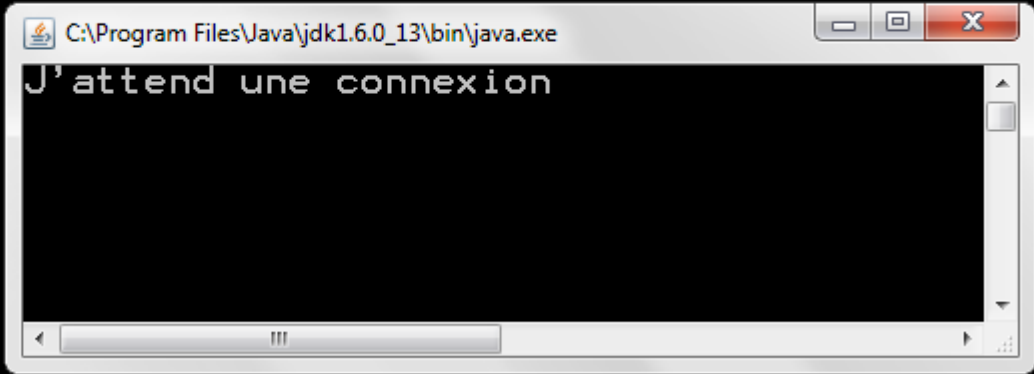
Implémentation d'un serveur multi-thread

- La classe qui gère les entrées-sorties : ServiceClient.java

```
import java.io.*; import java.net.*;
public class ServiceClient extends Thread {
    Socket client; int numero;
    public ServiceClient(Socket s,int num){client=s;numero=num; }
    public void run(){
        try {
            BufferedReader in=new BufferedReader(
                new InputStreamReader(client.getInputStream()));
            PrintWriter out=new PrintWriter(client.getOutputStream(),true);
            System.out.println("Client Num "+numero);
            out.println("Vous etes le client num "+numero);
            while(true){
                String s=in.readLine();
                out.println("Votre commande contient "+ s.length()+" caractères");
            } } catch (IOException e) { e.printStackTrace();}
    }
}
```


Lancement du serveur multithreads

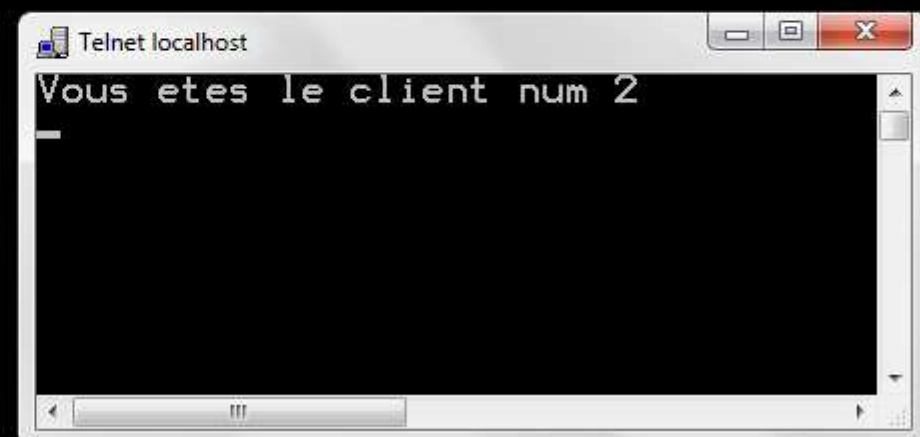
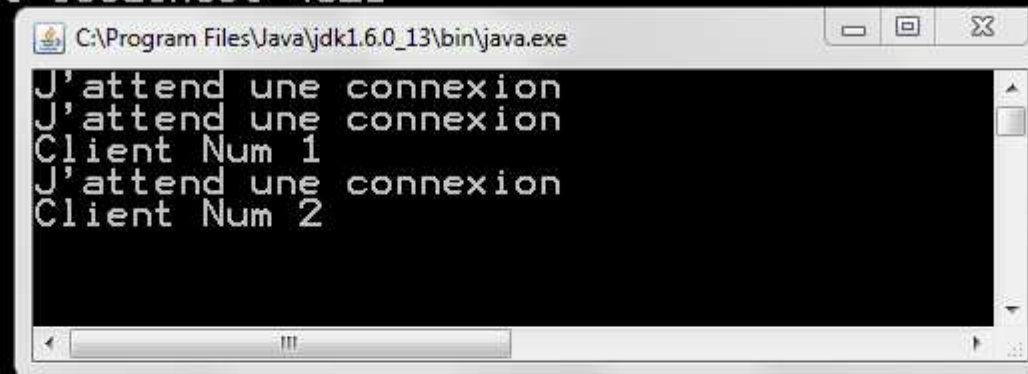
```
C:\JEEE\ASockets\bin>start java ServeurMultiThread  
C:\JEEE\ASockets\bin>
```



The screenshot shows a Windows command prompt window with a black background. The text 'C:\JEEE\ASockets\bin>start java ServeurMultiThread' is entered on the first line, and 'C:\JEEE\ASockets\bin>' is on the second line. Overlaid on this is a standard Windows application window titled 'C:\Program Files\Java\jdk1.6.0_13\bin\java.exe'. This window has a white title bar with minimize, maximize, and close buttons. The main content area is black and displays the text 'J'attends une connexion' in white. A vertical scrollbar is visible on the right side of the application window.

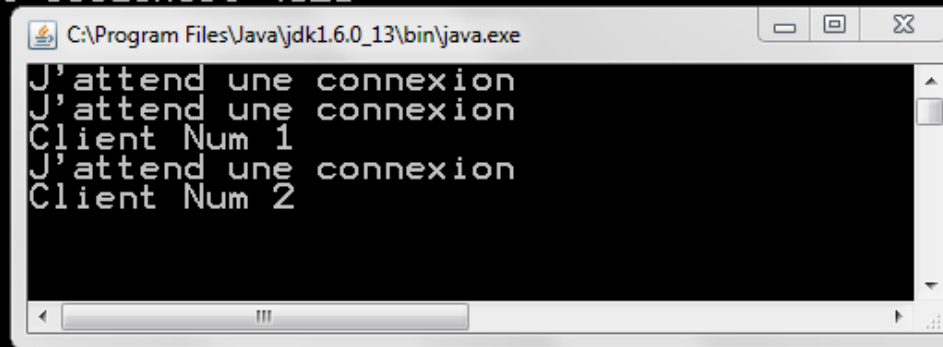
Lancement des clients avec telnet

```
C:\telnet>start telnet localhost 4321  
C:\telnet>start telnet localhost 4321  
C:\telnet>
```

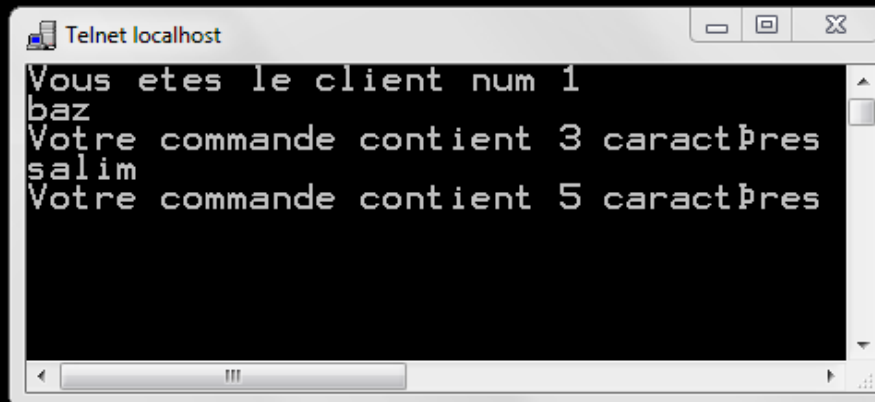


Communication entre les clients et le serveur

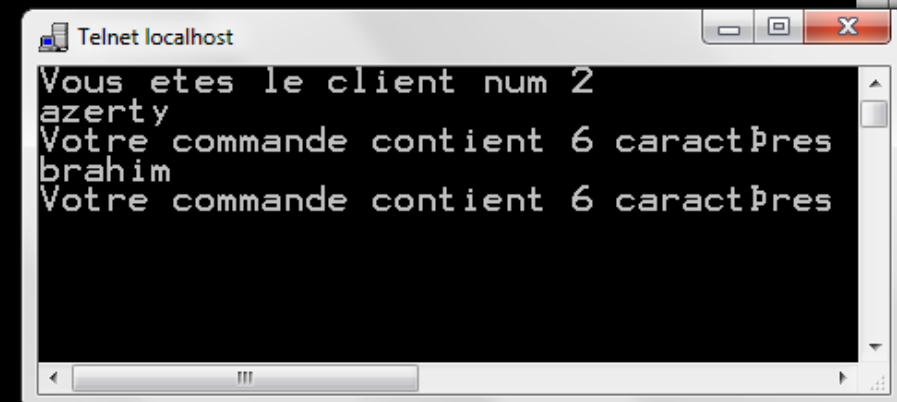
```
C:\telnet>start telnet localhost 4321  
C:\telnet>start telnet localhost 4321  
C:\telnet>
```



J'attends une connexion
J'attends une connexion
Client Num 1
J'attends une connexion
Client Num 2



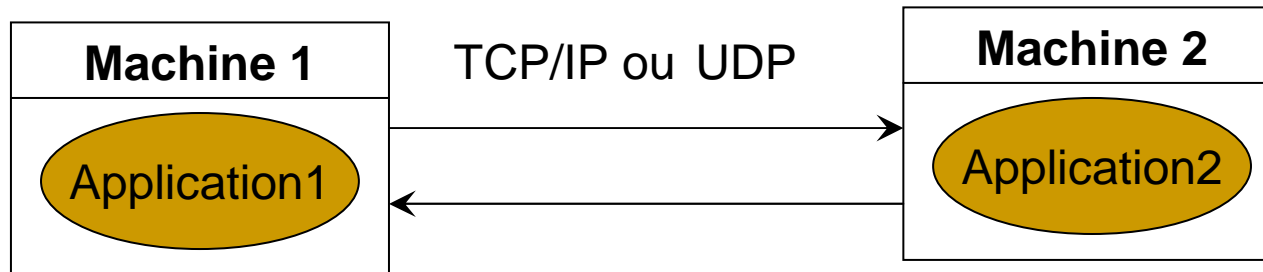
Telnet localhost
Vous etes le client num 1
baz
Votre commande contient 3 caractères
salim
Votre commande contient 5 caractères



Telnet localhost
Vous etes le client num 2
azerty
Votre commande contient 6 caractères
brahim
Votre commande contient 6 caractères

Architectures distribuées

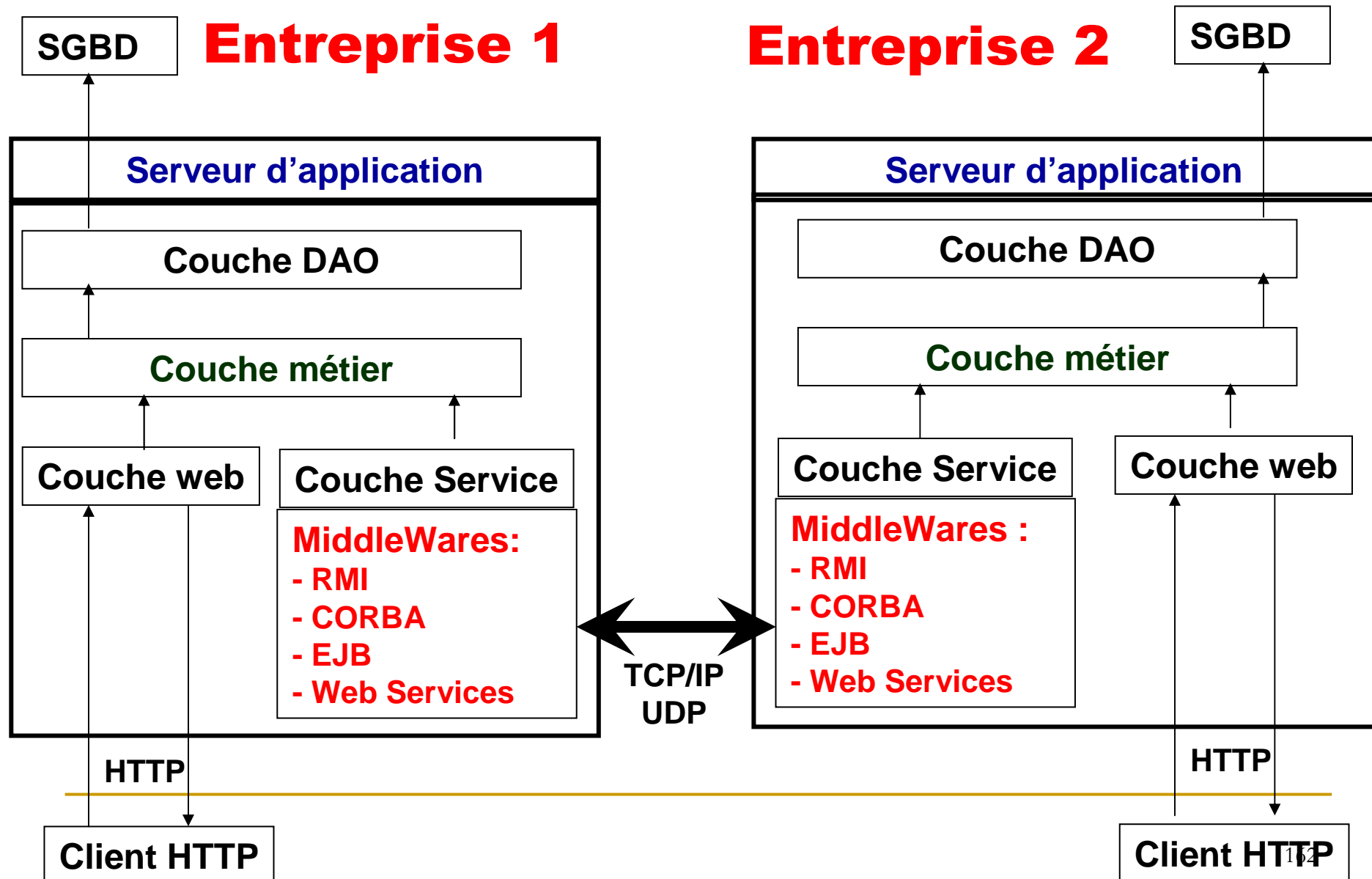
Applications distribuées



Technologies d'accès :

- Sockets
- RMI (JAVA)
- CORBA (Multi Langages)
- EJB (J2EE)
- Web Services (HTTP+XML)

Architectures Distribuées



Applications distribuées

- Une application distribuée est une application dont les classes sont réparties sur plusieurs machines différentes.
- Dans de telles applications, on peut invoquer des méthodes à distance.
- Il est alors possible d'appeler les méthode d'un objet qui n'est pas situé sur la machine locale.

Applications distribuées

■ RPC : Remote Procedure Calls

- ❑ Déjà dans le langage C, il était possible de faire de l'invocation à distance en utilisant RPC.
- ❑ RPC étant orienté "structure de données", il ne suit pas le modèle "orienté objet".

■ RMI : Remote Method Invocation

- ❑ RMI est un middleware qui permet l'utilisation d'objets sur des JVM distantes.
- ❑ RMI va plus loin que RPC puisqu'il permet non seulement l'envoi des données d'un objet, mais aussi de ses méthodes. Cela se fait en partie grâce à la sérialisation des objets
- ❑ RMI est utilisé pour créer des applications distribuées développées avec Java.

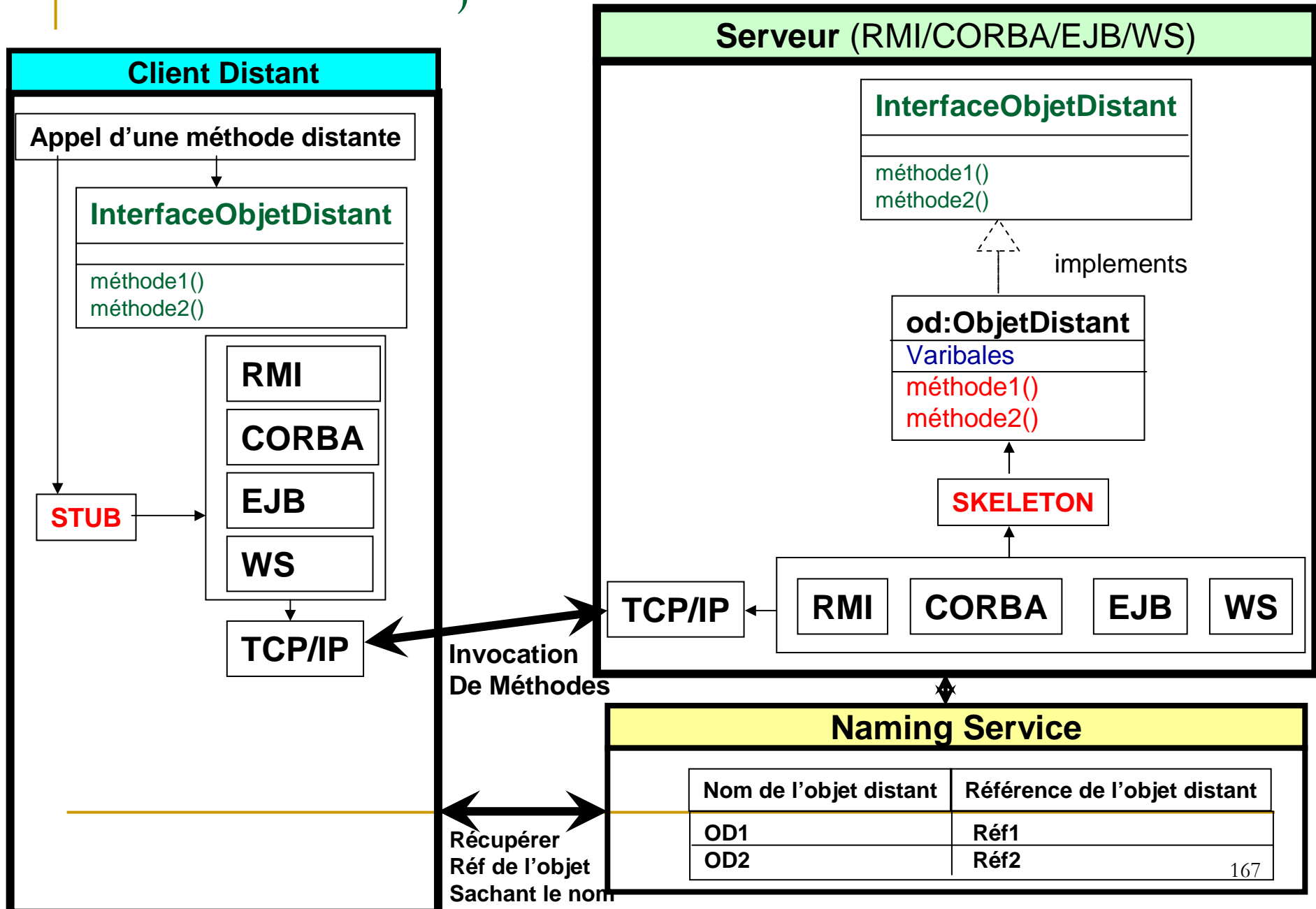
Application distribuée

- CORBA : Common Object Request Broker Architecture
 - Il existe une technologie, non liée à Java, appelée CORBA qui est un standard d'objet réparti.
 - CORBA a été conçue par l'OMG (Object Management Group), un consortium regroupant 700 entreprises dont Sun.
 - Le grand intérêt de CORBA est qu'il fonctionne sous plusieurs langages, mais il est plus lourd à mettre en place.

Application distribuée

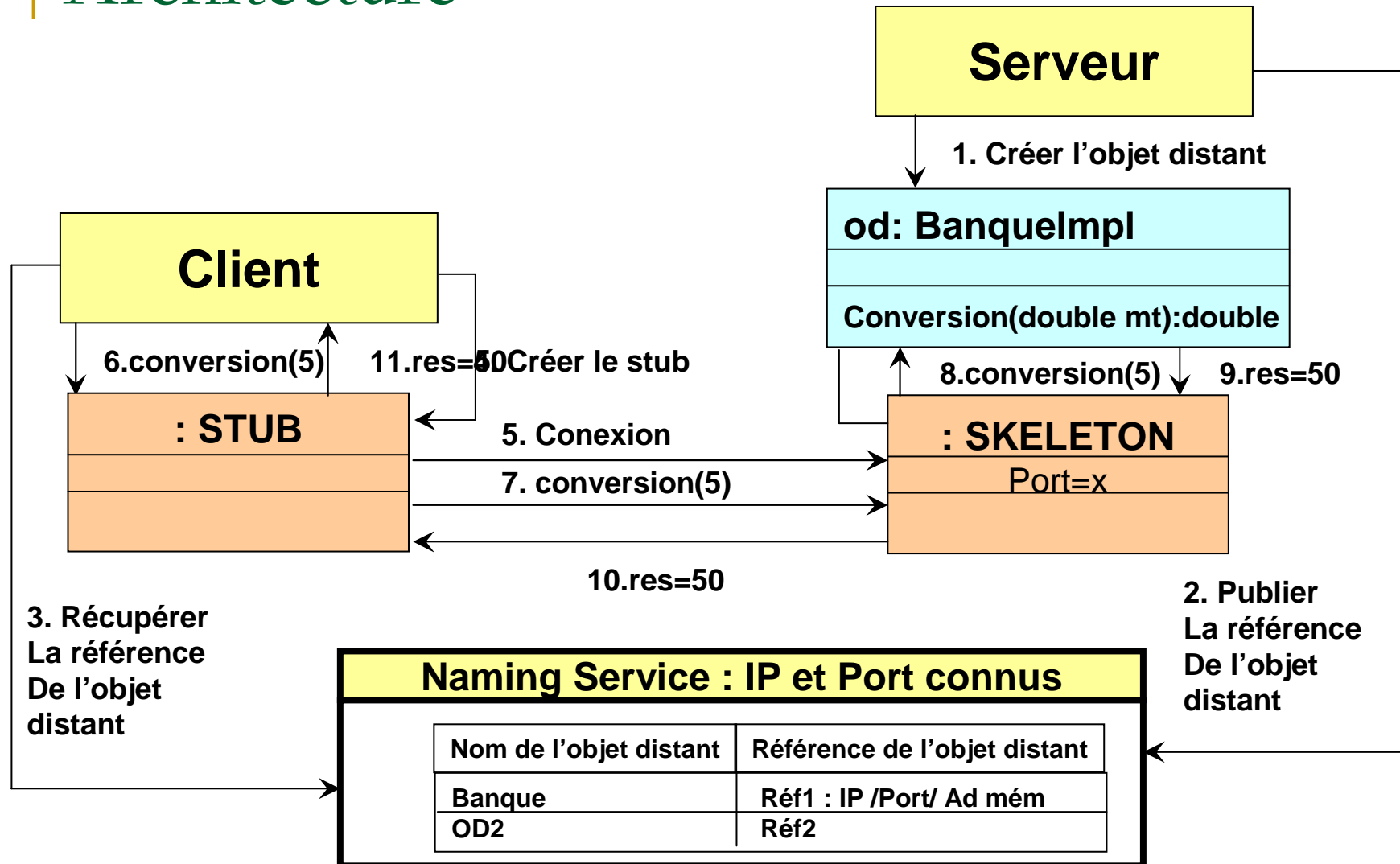
- EJB : Entreprise Java Beans
 - ❑ Avec Le développement des serveur d'applications J2EE Sun, a créé la technologie EJB qui permet également de créer des composants métier distribués.
 - ❑ Cette technologie est dédiée pour les applications J2EE
 - ❑ Les EJB utilise RMI et CORBA
- Web Services : (Protocole SOAP = HTTP + XML)
 - ❑ Avec le développement du web et de la technologie XML, La technologie de distribution « Web Services » a été développée.
 - ❑ Avec les web services, on peut développer des applications distribuées multi langages et multi plateformes.
 - ❑ Les web services sont plus facile à mettre en œuvre.
 - ❑ Les web services exploitent deux concepts XML et HTTP
- Toutes ces technologies de création des applications collaboratives s'appuient sur les SOCKETS

Accès à un objet distant via un middleware



RMI

Architecture



RMI

- Le but de RMI est de créer un modèle objet distribué Java
- RMI est apparu dès la version 1.1 du JDK
- Avec RMI, les méthodes de certains objets (appelés objets distants) peuvent être invoquées depuis des JVM différentes (espaces d'adressages distincts)
- En effet, RMI assure la communication entre le serveur et le client via TCP/IP et ce de manière transparente pour le développeur.
- Il utilise des mécanismes et des protocoles définis et standardisés tels que les sockets et RMP (Remote Method Protocol).
- Le développeur n'a donc pas à se soucier des problèmes de niveaux inférieurs spécifiques aux technologies réseaux.

Rappel sur les interfaces

- Dans java, une interface est une classe abstraite qui ne contient que des méthodes abstraites.
- Dans java une classe peut hériter d'une seule classe et peut implémenter plusieurs interfaces.
- Implémenter une interface signifie qu'il faut redéfinir toutes les méthodes déclarées dans l'interface.

Exemple

- Un exemple d'interface

```
public interface IBanque{  
    public void verser(float mt);  
}
```

- Un exemple d'implémentation

```
public class BanqueImpl implements IBanque{  
    private float solde ;  
  
    public void verser(float mt){  
        solde=solde+mt;  
    }  
}
```

Architecture de RMI

■ Interfaces:

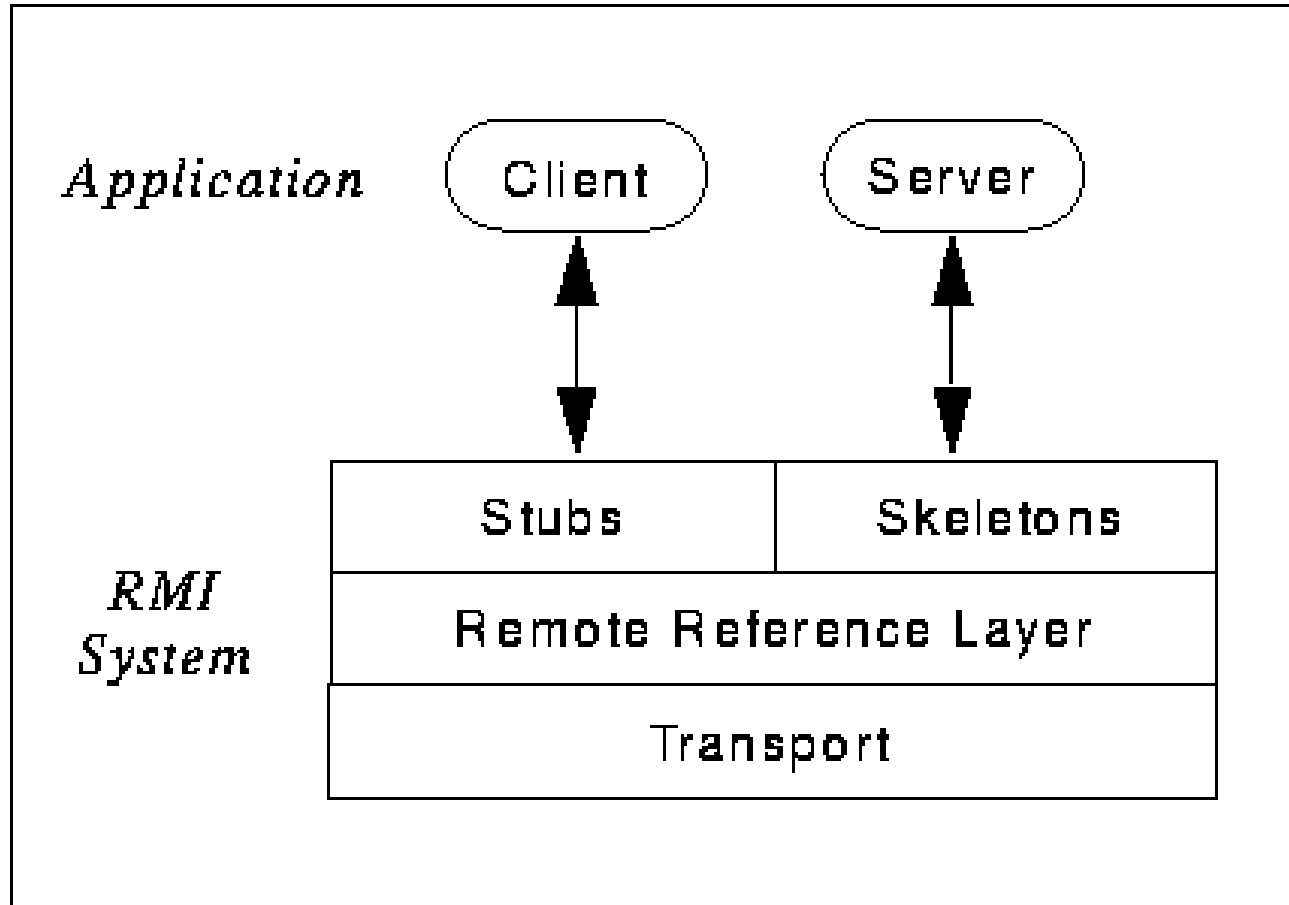
- ❑ Les interfaces est le cœur de RMI.
- ❑ L'architecture RMI est basé sur un principe important :
 - La définition du comportement et l'exécution de ce comportement sont des concepts séparés.
- ❑ La définition d'un service distant est codé en utilisant une interface Java.
- ❑ L'implémentation de ce service distant est codée dans une classe.

Coches de RMI

- RMI est essentiellement construit sur une abstraction en trois couches.
 - Stubs et Skeletons (Souche et Squelette)
 - Remote Reference Layer (Couche de référencement distante)
 - Couche Transport

Architecture de RMI

■ Architecture 1



Stubs et Skeletons

- Les stubs sont des classes placées dans la machine du client.
 - Lorsque notre client fera appel à une méthode distante, cet appel sera transféré au stub.
 - Le stub est donc un relais du côté du client. Il devra donc être placé sur la machine cliente.
 - Le stub est le représentant local de l'objet distant.
 - Il emballe les arguments de la méthode distante et les envoie dans un flux de données vers le service RMI distant.
 - D'autre part, il déballe la valeur ou l'objet de retour de la méthode distante.
 - Il communique avec l'objet distant par l'intermédiaire d'un skeleton.
-

Stubs et Skeletons

- Le squelette est lui aussi un relais mais du côté serveur. Il devra être placé sur la machine du serveur.
- Il débale les paramètres de la méthode distante, les transmet à l'objet local et embale les valeurs de retours à renvoyer au client.
- Les stubs et les skeletons sont donc des intermédiaires entre le client et le serveur qui gèrent le transfert distant des données.
- On utilise le compilateur `rmic` pour la génération des stubs et des skeletons avec le JDK
- Depuis la version 2 de Java, le skeleton n'existe plus. Seul le stub est nécessaire du côté du client mais aussi du côté serveur.

Remote Reference Layer

- La deuxième couche est la couche de référence distante.
- Ce service est assuré par le lancement du programme **rmiregistry** du côté du serveur
- Le serveur doit enregistrer la référence de chaque objet distant dans le service rmiregistry en attribuant un nom à cet objet distant.
- Du côté du client, cette couche permet l'obtention d'une référence de l'objet distant à partir de la référence locale (le stub) en utilisant son nom rmiregsitry

Couche transport

- La couche transport est basée sur les connexions TCP/IP entre les machines.
- Elle fournit la connectivité de base entre les 2 JVM.
- De plus, cette couche fournit des stratégies pour passer les firewalls.
- Elle suit les connexions en cours.
- Elle construit une table des objets distants disponibles.
- Elle écoute et répond aux invocations.
- Cette couche utilise les classes Socket et ServerSocket.
- Elle utilise aussi un protocole propriétaire R.M.P. (Remote Method Protocol).

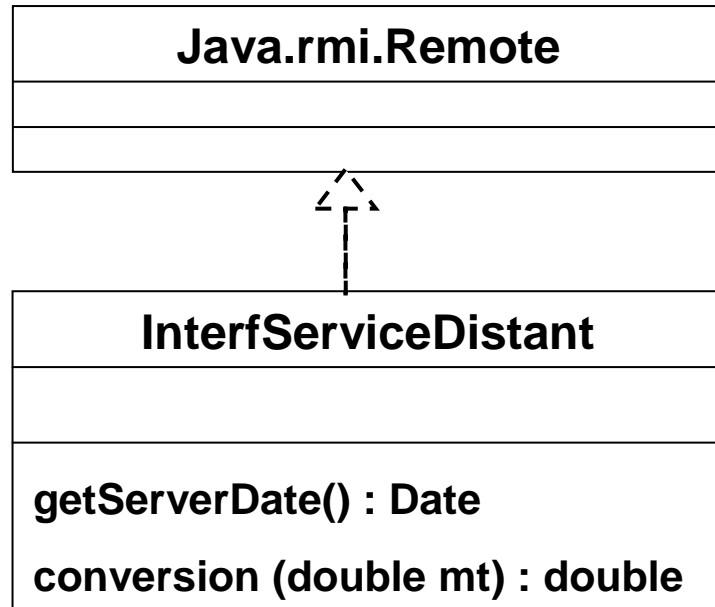
Démarche RMI

1. Créer les interfaces des objets distants
2. Créer les implémentations des objets distants
3. Générer les stubs et skeletons
4. Créer le serveur RMI
5. Créer le client RMI
6. Déploiement Lancement
 - Lancer l'annuaire RMIREGISTRY
 - Lancer le serveur
 - Lancer le client

Exemple

- Supposant qu'on veut créer un serveur RMI qui crée un objet qui offre les services distants suivant à un client RMI:
 - ❑ Convertir un montant de l'euro en DH
 - ❑ Envoyer au client la date du serveur.

1- Interface de l'objet distant



```
import java.rmi.Remote;import java.rmi.RemoteException;
import java.util.Date;
public interface InterfServiceDistant extends Remote {
    public Date getServerDate() throws RemoteException;
    public double convertEuroToDH(double montant) throws
RemoteException;
}
```

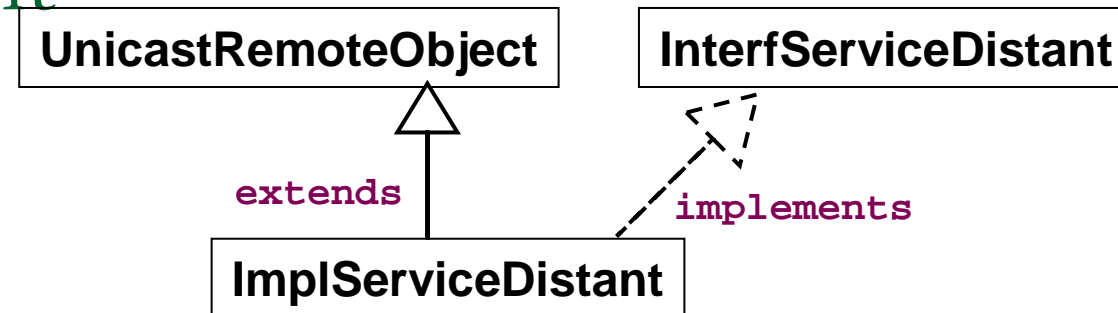
Interfaces

- La première étape consiste à créer une interface distante qui décrit les méthodes que le client pourra invoquer à distance.
- Pour que ses méthodes soient accessibles par le client, cette interface doit hériter de l'interface **Remote**.
- Toutes les méthodes utilisables à distance doivent pouvoir lever les exceptions de type **RemoteException** qui sont spécifiques à l'appel distant.
- Cette interface devra être placée sur les deux machines (serveur et client).

Implémentation

- Il faut maintenant implémenter cette interface distante dans une classe. Par convention, le nom de cette classe aura pour suffixe Impl.
- Notre classe doit hériter de la classe `java.rmi.server.RemoteObject` ou de l'une de ses sous-classes.
- La plus facile à utiliser étant `java.rmi.server.UnicastRemoteObject`.
- C'est dans cette classe que nous allons définir le corps des méthodes distantes que pourront utiliser nos clients

Deuxième étape : Implémentation de l'objet distant



```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class ImplServiceDistant extends UnicastRemoteObject implements
    InterfServiceDistant {
    public ImplServiceDistant() throws RemoteException{

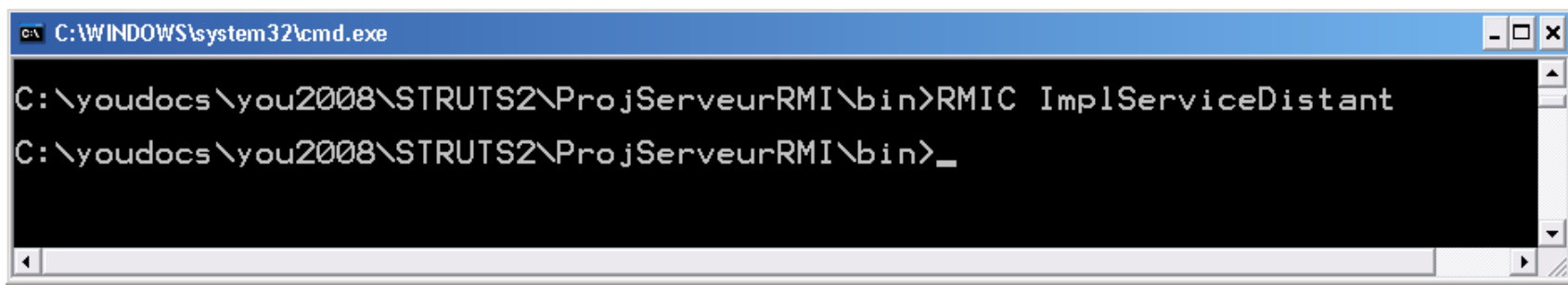
    }
    public Date getServerDate() throws RemoteException {
    return new Date();
    }
    public double convertEuroToDH(double montant) throws RemoteException {
    return montant*11.3;
    }
}
```

Implémentation

- Il faut maintenant implémenter cette interface distante dans une classe. Par convention, le nom de cette classe aura pour suffixe Impl.
- Notre classe doit hériter de la classe `java.rmi.server.RemoteObject` ou de l'une de ses sous-classes.
- La plus facile à utiliser étant `java.rmi.server.UnicastRemoteObject`.
- C'est dans cette classe que nous allons définir le corps des méthodes distantes que pourront utiliser nos clients

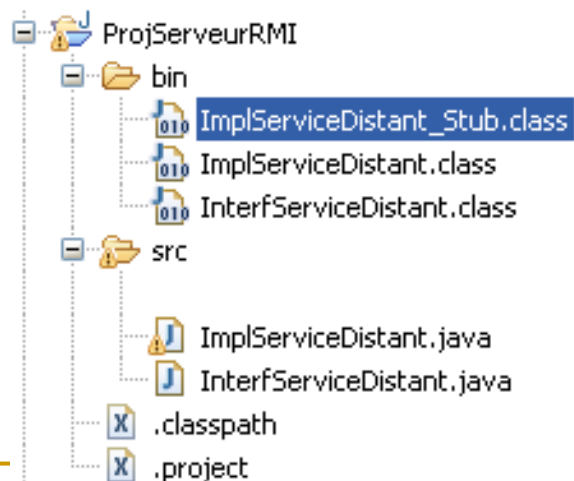
STUBS et SKELETONS

- Si l'implémentation est créée, les stubs et skeletons peuvent être générés par l'utilitaire rmic en écrivant la commande:
- Rmic NOM_IMPLEMENTATION



```
C:\WINDOWS\system32\cmd.exe

C:\youdocs\you2008\STRUTS2\ProjServeurRMI\bin>RMIC ImplServiceDistant
C:\youdocs\you2008\STRUTS2\ProjServeurRMI\bin>_
```



Naming Service

- Les clients trouvent les services distants en utilisant un service d'annuaire activé sur un hôte connu avec un numéro de port connu.
- RMI peut utiliser plusieurs services d'annuaire, y compris Java Naming and Directory Interface (JNDI).
- Il inclut lui-même un service simple appelé (rmiregistry).
- Le registre est exécuté sur chaque machine qui héberge des objets distants (les serveurs) et accepte les requêtes pour ces services, par défaut sur le port 1099.

Utilisation de RMI

- Un serveur crée un service distant en créant d'abord un objet local qui implémente ce service.
- Ensuite, il exporte cet objet vers RMI.
- Quand l'objet est exporté, RMI crée un service d'écoute qui attend qu'un client se connecte et envoie des requêtes au service.
- Après exportation, le serveur enregistre cet objet dans le registre de RMI sous un nom public qui devient accessible de l'extérieur.
- Le client peut alors consulter le registre distant pour obtenir des références à des objets distants.

Le serveur

- Notre serveur doit enregistrer auprès du registre RMI l'objet local dont les méthodes seront disponibles à distance.
- Cela se fait grâce à la méthode statique **bind()** ou **rebind()** de la classe Naming.
- Cette méthode permet d'associer (enregistrer) l'objet local avec un synonyme dans le registre RMI.
- L'objet devient alors disponible par le client.
 - ❑ `ObjetDistantImpl od = new ObjetDistantImpl();`
 - ❑ `Naming.rebind("rmi://localhost:1099/NOM_Service",od);`

Code du serveur RMI

```
import java.rmi.Naming;
public class ServeurRMI {
    public static void main(String[] args) {
        try {
            // Créer l'objet distant
            ImplServiceDistant od=new ImplServiceDistant();
            // Publier sa référence dans l'annuaire
            Naming.rebind("rmi://localhost:1099/SD",od);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Client

- Le client peut obtenir une référence à un objet distant par l'utilisation de la méthode statique **lookup()** de la classe Naming.
- La méthode lookup() sert au client pour interroger un registre et récupérer un objet distant.
- Elle retourne une référence à l'objet distant.
- La valeur retournée est du type Remote. Il est donc nécessaire de caster cet objet en l'interface distante implémentée par l'objet distant.

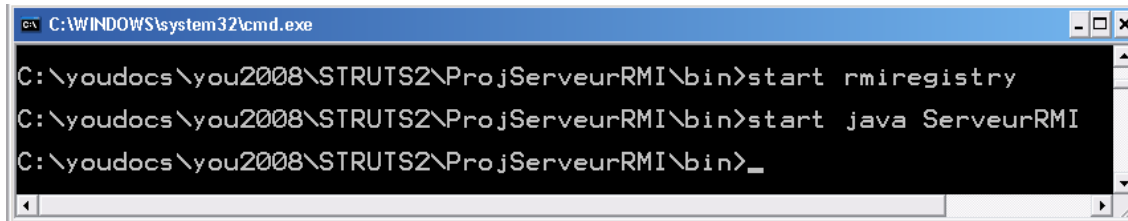
```
import java.rmi.Naming;
public class ClientRMI {
    public static void main(String[] args) {
        try {
            InterfServiceDistant stub=
                (InterfServiceDistant)Naming.lookup("rmi://localhost:1099/SD");
            System.out.println("Date du serveur:"+stub.getServerDate());
            System.out.println("35 euro vaut "+stub.convertEuroToDH(35));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Lancement

- Il est maintenant possible de lancer l'application. Cela va nécessiter l'utilisation de trois consoles.
 - ❑ La première sera utilisée pour activer le registre. Pour cela, vous devez exécuter l'utilitaire **rmiregistry**
 - ❑ Dans une deuxième console, exécutez le serveur. Celui-ci va charger l'implémentation en mémoire, enregistrer cette référence dans le registre et attendre une connexion cliente.
 - ❑ Vous pouvez enfin exécuter le client dans une troisième console.
- Même si vous exécutez le client et le serveur sur la même machine, RMI utilisera la pile réseau et TCP/IP pour communiquer entre les JVM.

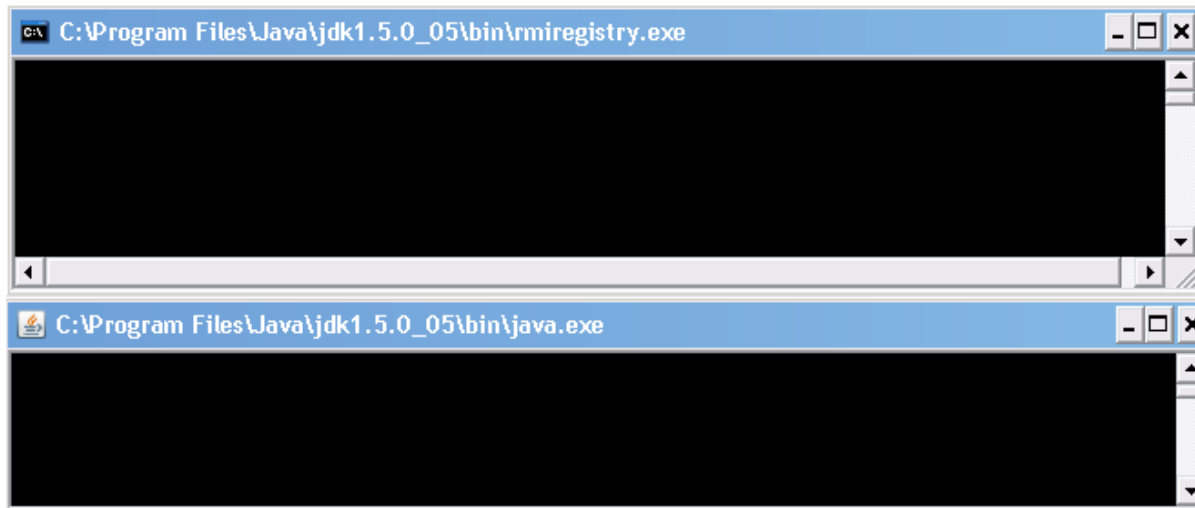
Lancement

Lancement de l'annuaire puis du serveur rmi :

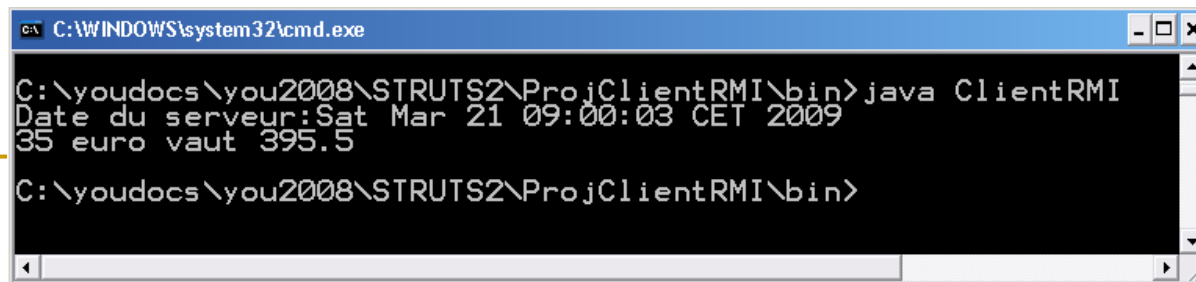


```
C:\WINDOWS\system32\cmd.exe

C:\youdocs\you2008\STRUTS2\ProjServeurRMI\bin>start rmiregistry
C:\youdocs\you2008\STRUTS2\ProjServeurRMI\bin>start java ServeurRMI
C:\youdocs\you2008\STRUTS2\ProjServeurRMI\bin>_
```



Lancement du client RMI :



```
C:\WINDOWS\system32\cmd.exe

C:\youdocs\you2008\STRUTS2\ProjClientRMI\bin>java ClientRMI
Date du serveur:Sat Mar 21 09:00:03 CET 2009
35 euro vaut 395.5
C:\youdocs\you2008\STRUTS2\ProjClientRMI\bin>
```

JNDI (Java Naming and Directory Interface)

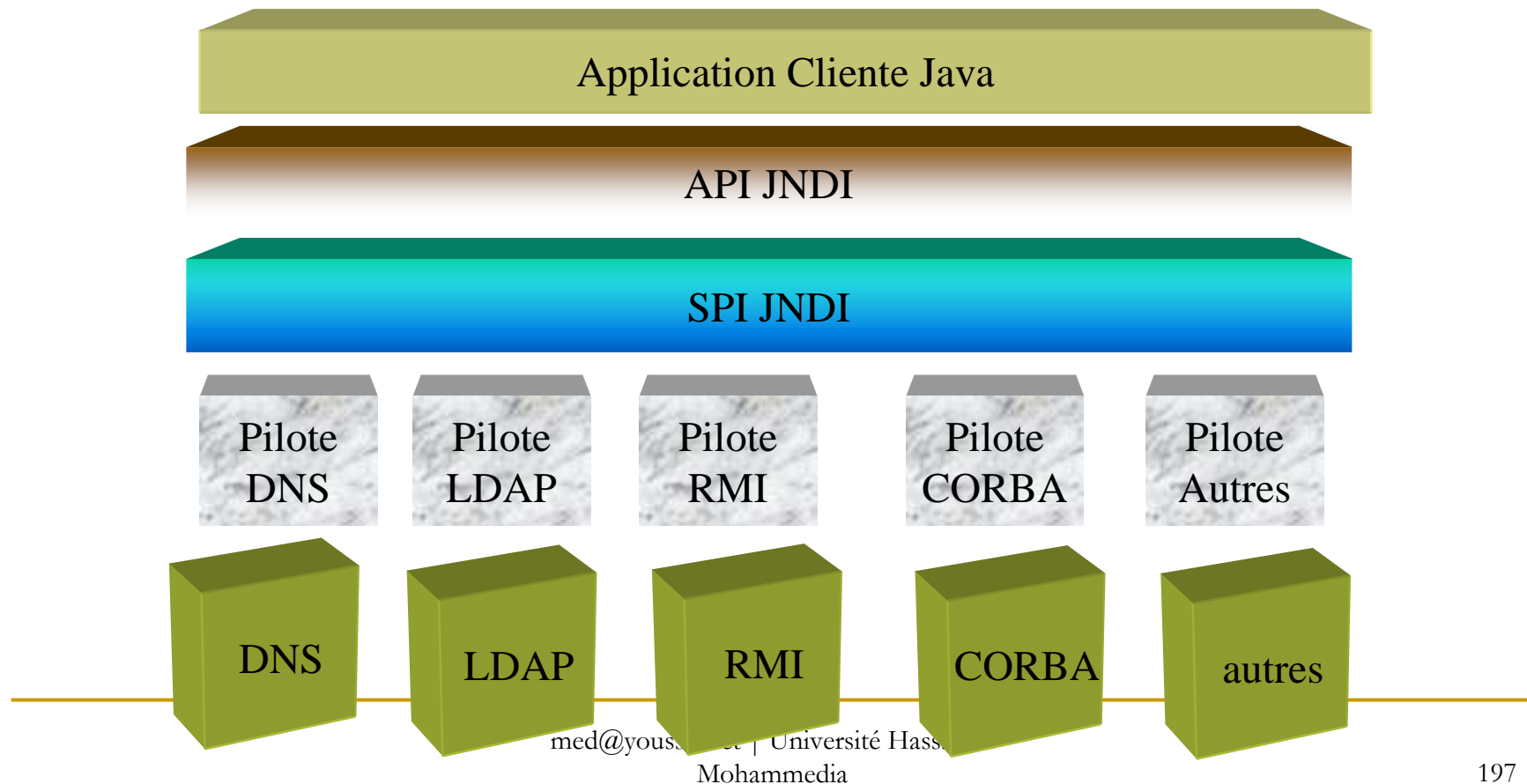
- JNDI est l'acronyme de Java Naming and Directory Interface.
- Cette API fournit une interface unique pour utiliser différents services de nommages ou d'annuaires et définit une API standard pour permettre l'accès à ces services.
- Il existe plusieurs types de service de nommage parmi lesquels :
 - DNS (Domain Name System) : service de nommage utilisé sur internet pour permettre la correspondance entre un nom de domaine et une adresse IP
 - LDAP(Lightweight Directory Access Protocol) : annuaire
 - NIS (Network Information System) : service de nommage réseau développé par Sun Microsystems
 - COS Naming (Common Object Services) : service de nommage utilisé par Corba pour stocker et obtenir des références sur des objets Corba
 - RMI Registry : service de nommage utilisé par RMI
 - etc, ...

JNDI (Java Naming and Directory Interface)

- Un service de nommage permet d'associer un nom unique à un objet et de faciliter ainsi l'obtention de cet objet.
- Un annuaire est un service de nommage qui possède en plus une représentation hiérarchique des objets qu'il contient et un mécanisme de recherche.
- JNDI propose donc une abstraction pour permettre l'accès à ces différents services de manière standard. Ceci est possible grâce à l'implémentation de pilotes qui mettent en œuvre la partie SPI de l'API JNDI. Cette implémentation se charge d'assurer le dialogue entre l'API et le service utilisé.

Architecture JNDI

- L'architecture de JNDI se compose d'une API et d'un Service Provider Interface (SPI). Les applications Java emploient l'API JNDI pour accéder à une variété de services de nommage et d'annuaire. Le SPI permet de relier, de manière transparente, une variété de services de nommage et d'annuaire ; permettant ainsi à l'application Java d'accéder à ces services en utilisant l'API JNDI



Architecture JNDI

- JNDI est inclus dans le JDK Standard (Java 2 SDK) depuis la version 1.3. Pour utiliser JNDI, vous devez posséder les classes JNDI et au moins un SPI (JNDI Provider).
- La version 1.4 du JDK inclut 3 SPI pour les services de nommage/annuaire suivant :
 - ❑ Lightweight Directory Access Protocol (LDAP)
 - ❑ Le service de nommage de CORBA (Common Object Request Broker Architecture) Common Object Services (COS)
 - ❑ Le registre de RMI (Remote Method Invocation) rmiregistry
 - ❑ DNS

Serveur RMI utilisant JNDI

```
package service; import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;
public class ServeurRMI {
public static void main(String[] args) {
try {
    // Démarrer l'annuaire RMI REgistry
    LocateRegistry.createRegistry(1099);
    // Créer l'Objet distant
    BanqueImpl od=new BanqueImpl();
    // Afficher la référence de l'objet distant
    System.out.println(od.toString());
    // Créer l'objet InitialContext JNDI en utilisant le fichier
    jndi.properties
    Context ctx=new InitialContext();
    // Publier la référence de l'objet distant avec le nom BK
    ctx.bind("SD", od);
} catch (Exception e) { e.printStackTrace();}
}
}
```

Fichier jndi.properties :

```
java.naming.factory.initial=com.sun.jndi.rmi.registry.RegistryContextFactory
java.naming.provider.url=rmi://localhost:1099
```

Client RMI utilisant JNDI

```
import javax.naming.*;
import rmi.IBanque;
public class ClientRMI {
public static void main(String[] args) {
try {
    Context ctx=new InitialContext();
    ctx.addToEnvironment(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.rmi.registry.RegistryContextFactory");
    ctx.addToEnvironment("java.naming.provider.url",
        "rmi://localhost:1099");
    IBanque stub=(IBanque) ctx.lookup("BK");
    System.out.println(stub.test("test"));
} catch (Exception e) {
e.printStackTrace();
}}}
```

- Dans ce cas, on pas besoin de créer le fichier jndi.properties