



Socket Java

Communication à distance et synchronisation



- Echanges entre processus
 - synchrones → bloquants
 - asynchrones → non bloquants
- Communications synchrones
 - émetteur envoie un message et attend son accusé de réception
 - récepteur attend jusqu'à la réception d'un message
- Communications asynchrones
 - émetteur envoie un message et continue immédiatement
 - récepteur n'est pas bloqué → utilisation d'un système d'interruption ou de scrutation pour traiter le message reçu



Java utilise une émission asynchrone et une réception synchrone

Les Sockets

- C'est le mécanisme sous-jacent pour toute communication portable sur un réseau
 - Client/serveur
- Un **socket** est un moyen de communication entre deux processus
 - Interface de connexion
- Interface de bas niveau de programmation pour la communication réseau
 - échange de flots de données entre des applications qui ne sont pas forcément sur la même machine.
- La majorité des E/S en Java (dont les E/S réseaux) utilisent les classes de Streams.
 - Lire/écrire sur le réseau <=> Lire/écrire dans un fichier

Petit rappel

ATTENTION!



Client : celui qui initie la communication

Serveur : celui qui prend en charge la requête

- Port : canal dédié à un service
 - 80 pour http
 - 21 pour ftp
 - 23 pour telnet
 - ...

Les ports de 1 à 1024 sont réservés aux processus systèmes et services standards

Les Sockets



- Dans le package `java.net`
- Deux classes principales
 - La classe *DatagramSocket* utilise UDP
 - La classe *Socket* utilise le protocole TCP

La classe *DatagramSocket*



- *DatagramSocket* utilise le protocole UDP
 - UDP : *User Datagram Protocol* (*couche transport*)
- *C'est un peu comme les cartes postales*

Une application peut envoyer un court message à d'autres applications, mais rien n'est fait pour

- conserver une liaison entre 2 messages
- s'assurer que les messages arrivent dans l'ordre
- faire en sorte qu'aucun message ne soit perdu



UDP est un protocole dit *non-reliable*

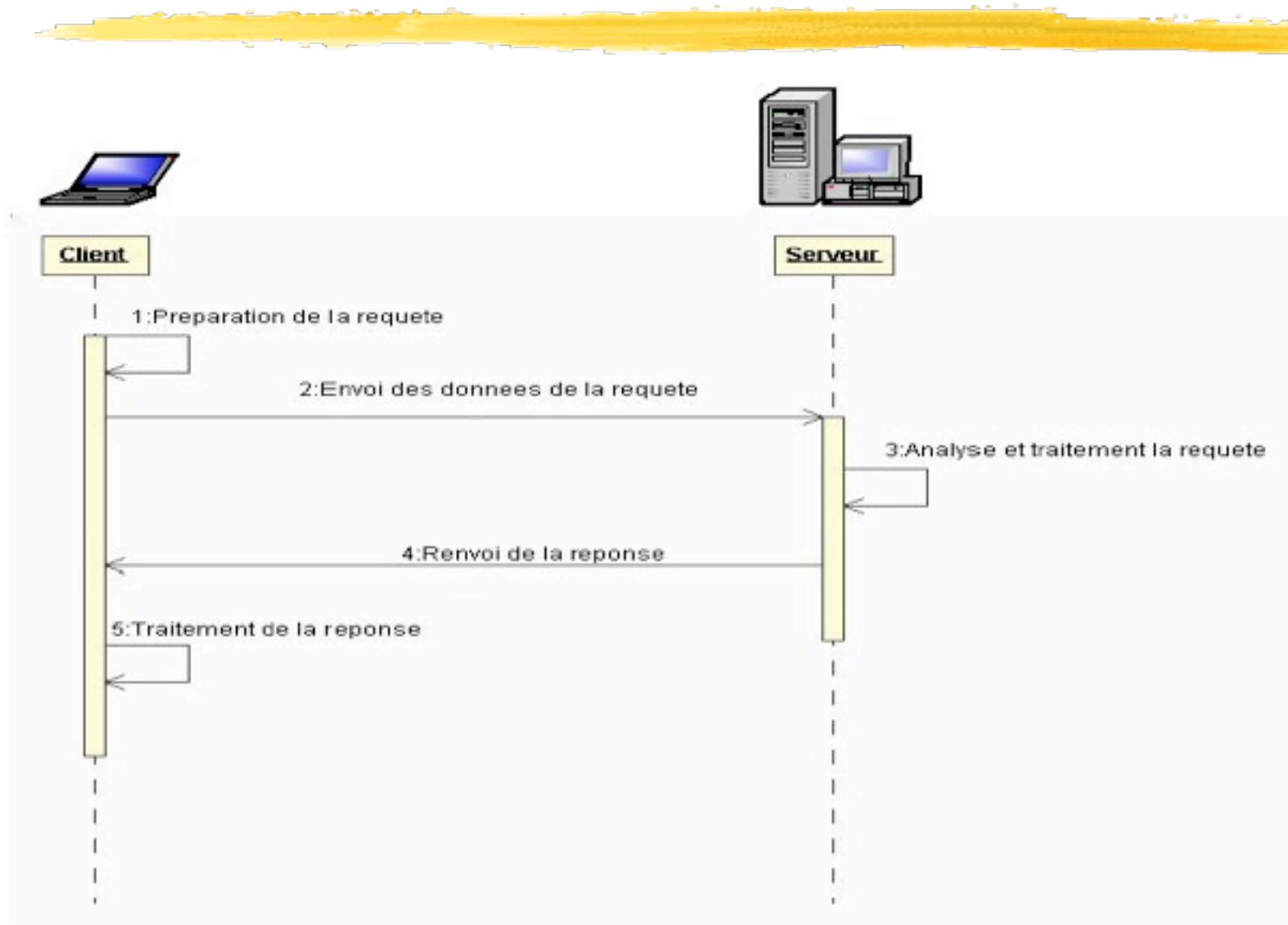
La classe *Socket*

- ***Socket* utilise le protocole TCP**
 - TCP : *Transmission Control Protocol* (*couche transport*)
- ***C'est un peu comme le téléphone***
 - Après avoir établi la connexion, 2 applications peuvent s'envoyer des données
 - La connexion reste en place même quand personne ne parle
 - Le protocole s'assure qu'aucune donnée n'est perdue
 - Les données arrivent toujours dans l'ordre où elles ont été envoyées

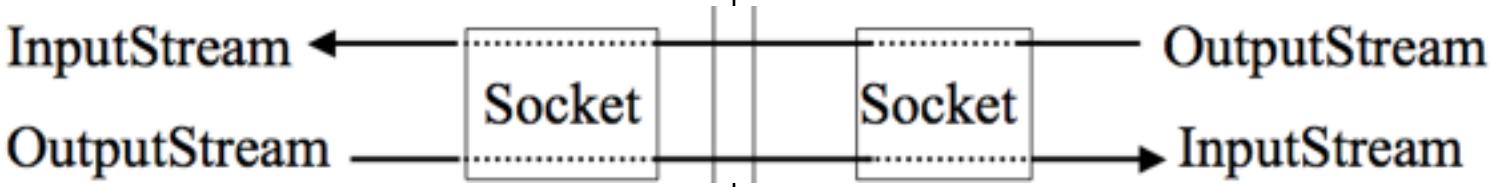


TCP est un protocole dit *reliable*

Le modèle client-serveur Java



Le modèle client-serveur Java

Serveur	Client
1- Enregister le service: ServerSocket(port,#nb_cnx)	1 - Etablir la connexion: Socket(host,port) ---> création d'un objet Socket
2- Attendre une connexion client accept() --> retourne un objet Socket	2- Utiliser le socket
3 - Utiliser le socket	
	
close()	close

Un serveur TCP/IP



- il utilise la classe `java.net.ServerSocket` pour accepter des connexions de clients
- La méthode `accept()` retourne un objet de la classe `Socket` lorsqu'un client veut se connecter
 - ▶ quand un client se connecte à un port sur lequel un `ServerSocket` écoute, `ServerSocket` crée une nouvelle instance (objet) de la classe `Socket`. C'est grâce à cet objet que l'on pourra établir une connexion avec le client

```
int port = ...;
ServerSocket server = new ServerSocket(port);
Socket connection = server.accept();
```

Un serveur TCP/IP

- les constructeurs et la plupart des méthodes peuvent générer une IOException
- la méthode accept() est dite bloquante.

```
final int PORT = ...;

try {
    ServerSocket serveur = new ServerSocket(PORT,5);
    while (true) {
        Socket socket = serveur.accept();
    }
}
catch (IOException e){
    ....
}
```

Un client TCP/IP

- le client se connecte au serveur en créant une instance de la classe `java.net.Socket` : connexion synchrone

```
String host = ...;
int port = ...;
Socket connection = new Socket (host,port);
```

- le socket permet de supporter les communications côté client
- la méthode `close()` ferme (détruit) le socket
- les constructeurs et la plupart des méthodes peuvent générer une `IOException`
- le serveur doit être démarré avant le client. Dans le cas contraire, si le client se connecte à un serveur inexistant, une exception sera levée après un time-out

Un client TCP/IP

- Un serveur peut traiter simultanément plusieurs communications
 - ▶ Un seul objet ServerSocket
 - ▶ Un objet Socket par client
- Un client a besoin de deux informations pour trouver une machine serveur sur Internet et pour s'y connecter :
 - ▶ un nom de machine (hostname)
 - ▶ un numéro de port

```
final String HOST = "...";  
final int PORT = ...;  
  
try {  
    Socket socket = new Socket(HOST,PORT);  
}  
finally {  
    try {socket.close();} catch (IOException e){}  
}
```

Flux de données

```
try {  
    Socket socket = new Socket(HOST,PORT);  
    //Lecture du flux d'entrée en provenance du serveur  
    InputStreamReader reader = new  
        InputStreamReader(socket.getInputStream());  
    BufferedReader istream = new BufferedReader(reader);  
    String line = istream.readLine();  
    //Echo la ligne lue vers le serveur  
    PrintWriter ostream = new PrintWriter(socket.getOutputStream());  
    ostream.println(line);  
    ostream.flush();  
} catch (IOException e) {...}  
finally {try{socket.close();} catch (IOException e){}}
```

Un serveur TCP/IP multiclients

- le serveur précédent accepte plusieurs connexions simultanées, mais ne traite qu'un client à la fois, les autres sont mis en attente pour y remédier, utiliser les threads java (java.lang.Thread)

```
try{  
    ServerSocket serveur = new ServerSocket(PORT);  
    while (true) {  
        //accepter une connexion  
        Socket socket = serveur.accept();  
        // créer un thread : pour échanger les données avec le client  
        Connexion c = new Connexion(socket);  
        Thread processus_connexion = new Thread(c);  
        processus_connexion.start();  
    }  
} catch (IOException e) {...}
```

```
class Connexion implements Runnable  
{.....  
public Connexion (Socket s) {  
    this.s = s;  
    try{  
        in=new BufferedReader(  
                new  
InputStreamReader(s.getInputStream()));  
  
        out = new PrintWriter(s.getOutputStream());  
    } catch (IOException e) {...}  
public void run() {  
    try{  
        while (true) {  
            String ligne = in.readLine();  
            if (ligne == null) break; //fin de connexion c  
            output.println(ligne); out.flush();  
        }  
    } catch (IOException e) {...}  
    finally {try{  
s.close();}catch (IOException e){}}}
```

Un socket UDP



- Classes utilisées pour communication via UDP
 - ▶ InetAddress : codage des adresses IP
 - ▶ DatagramSocket : socket mode non connecté (UDP)
 - ▶ DatagramPacket : paquet de données envoyé via une socket sans connexion (UDP)

Classe InetAddress



- Méthodes:

- ▶ `public static InetAddress getByName(String host) throws UnknownHostException`

Crée un objet InetAddress identifiant une machine dont le nom est passé en paramètre

- ▶ `public static InetAddress getLocalHost() throws UnknownHostException`

Retourne l'adresse IP de la machine sur laquelle tourne le programme, c'est-à-dire l'adresse IP locale

Classe DatagramPacket

- Constructeurs

- ▶ `public DatagramPacket(byte[] buf, int length)`

- ✓ Création d'un paquet pour recevoir des données (sous forme d'un tableau d'octets)
 - ✓ Les données reçues seront placées dans buf
 - ✓ length précise la taille max de données à lire
 - ✓ Variante du constructeur : avec un offset pour ne pas commencer au début du tableau

- ▶ `public DatagramPacket(byte[] buf, int length, InetAddress address, int port)`

- ✓ Création d'un paquet pour envoyer des données (sous forme d'un tableau d'octets)
 - ✓ buf : contient les données à envoyer
 - ✓ length : longueur des données à envoyer
 - ✓ address : adresse IP de la machine destinataire des données
 - ✓ port : numéro de port distant (sur la machine destinataire) où envoyer les données

Classe DatagramPacket



- Méthodes

- ▶ `InetAddress getAddress()`

- ✓ Si paquet à envoyer : adresse de la machine destinataire
 - ✓ Si paquet reçu : adresse de la machine qui a envoyé le paquet

- ▶ `int getPort()`

- ▶ `byte[] getData`

- ▶ `int getLength()`

- ▶ `void setAddress(InetAddress adr)`

- ▶ `void setPort(int port)`

- ✓ Positionne l'adresse IP et le numéro de port de la machine destinataire du paquet

- ▶ `void setData(byte[] data)`

- ▶ `int setLength(int length)`

- ✓ Positionne les données à envoyer et sa longueur

Classe DatagramSocket



- Constructeurs

- ▶ `public DatagramSocket() throws SocketException`

- ✓ Crée une nouvelle socket en la liant à un port quelconque libre
 - ✓ Exception levée en cas de problème (a priori il doit pas y en avoir)

- ▶ `public DatagramSocket(int port) throws SocketException`

- ✓ Crée une nouvelle socket en la liant au port local précisé par le paramètre port
 - ✓ Exception levée en cas de problème : notamment quand le port est déjà occupé

Classe DatagramSocket

- Méthodes

- ▶ `public void send(DatagramPacket p) throws IOException`

- ✓ Envoie le paquet passé en paramètre. Le destinataire est identifié par le couple @IP/port précisé dans le paquet

- ▶ `public void receive(DatagramPacket p) throws IOException`

- ✓ Reçoit un paquet de données

- ✓ Bloquant tant qu'un paquet n'est pas reçu

- ✓ Quand paquet arrive, les attributs de p sont modifiés

- ✓ Les données reçues sont copiées dans le tableau passé en paramètre lors de la création de p et sa longueur est positionnée avec la taille des données reçues

- ✓ Les attributs d'@IP et de port de p contiennent l'@IP et le port de la socket distante qui a émis le paquet

- ▶ `public void close()`

- ✓ Ferme la socket et libère le port à laquelle elle était liée

Un socket UDP

```
//Machine destinataire
InetAddress address = InetAddress.getByName(" rainbow.essi.fr");
static final int PORT = 4562;
//Création du message à envoyer
String s = new String (" Message à envoyer");
int longueur = s.length();
byte[] message = new byte[longueur];
s.getBytes(0,longueur,message,0);
//Initialisation du paquet avec toutes les informations
DatagramPacket paquet = new DatagramPacket(message,longueur,
                                              address,PORT);
//Création du socket et envoi du paquet
DatagramSocket socket = new DatagramSocket();
socket.send(paquet);....
```

Un socket UDP

```
//Définir un buffer de réception
byte[] buffer = new byte[1024];
//On associe un paquet à un buffer vide pour la réception
DatagramPacket paquet =new
    DatagramPacket(buffer,buffer.length());
//On crée un socket pour écouter sur le port
DatagramSocket socket = new DatagramSocket(PORT);
while (true) {
//attente de réception
socket.receive(paquet);
//affichage du paquet reçu
String s = new String(buffer,0,0,paquet.getLength());
System.out.println("Paquet reçu : " + s);
```

R.M.I



Les objets distribués en Java

Le rêve de tout système distribué

L'idéal serait d'avoir un système distribué utilisant la technologie objet et permettant :

1) d'invoquer une méthode d'un objet se trouvant sur une autre machine exactement de la même manière que s'il se trouvait au sein de la même machine :

```
objetDistant.methode() ;
```

2) d'utiliser un objet distant (OD), sans savoir où se il se trouve, en demandant à un service « dédié » de renvoyer son adresse :

```
objetDistant =  
ServiceDeNoms.recherche(«monObjet»);
```

Le rêve de tout système distribué (suite)

3) de pouvoir passer un OD en paramètre d'appel à une méthode locale ou distante :

```
Resultat = objetLocal.methode(objetDistant);
```

```
Resultat = objetDistant.methode(autreObjetDistant);
```

4) de pouvoir récupérer le résultat d'un appel distant sous forme d'un nouvel objet qui aurait été créé sur la machine distante :

```
AutreObjetDistant = ObjetDistant.methode();
```

Objets Distsants

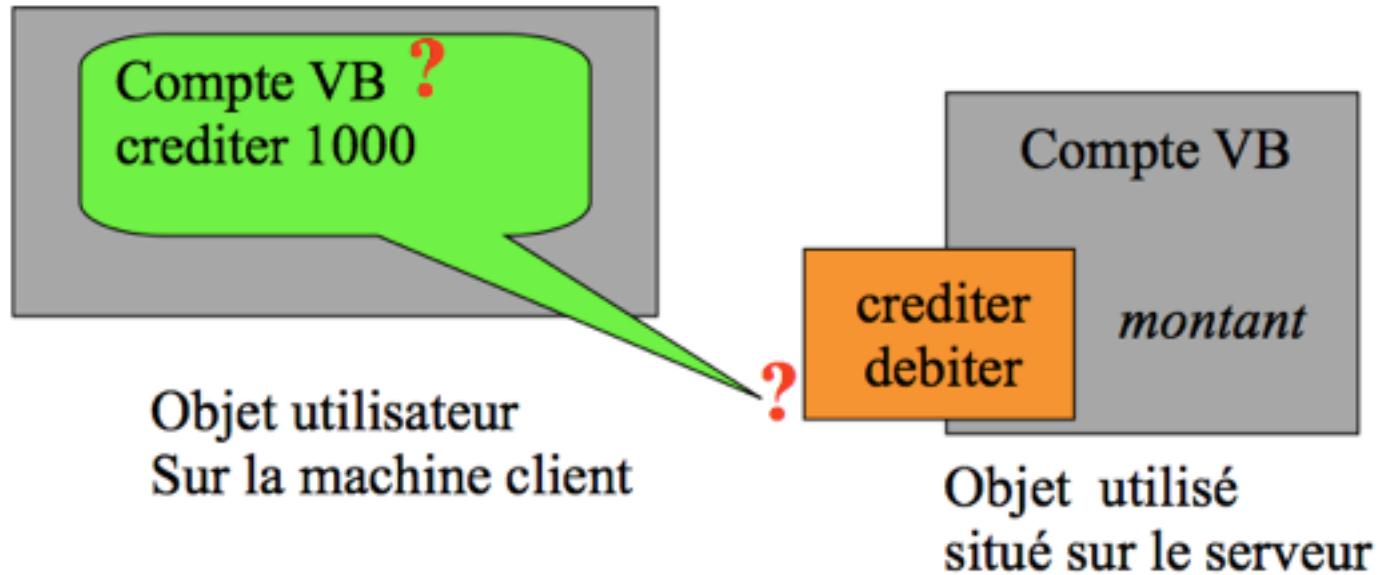


Un Objet Distant (OD) :

est un objet situé sur une machine distante de celle voulant utiliser l'objet.

Le terme « distant » signifie que l'objet est utilisé par l'intermédiaire d'un type spécial d'objet interface qui transfère les invocations de méthodes, paramètres et valeurs de retour sur le réseau.

Utilisation d'objets distants

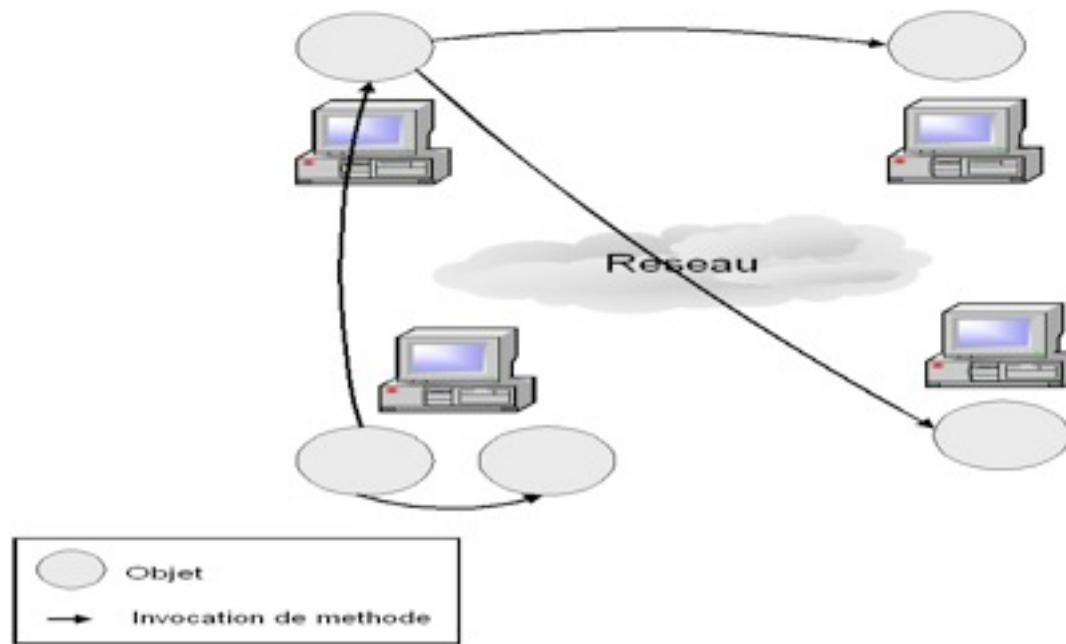


crediter et débiter <-> services (méthodes) proposés par le serveur

Un serveur peut abriter plusieurs objets distants et plusieurs types d'objets distants

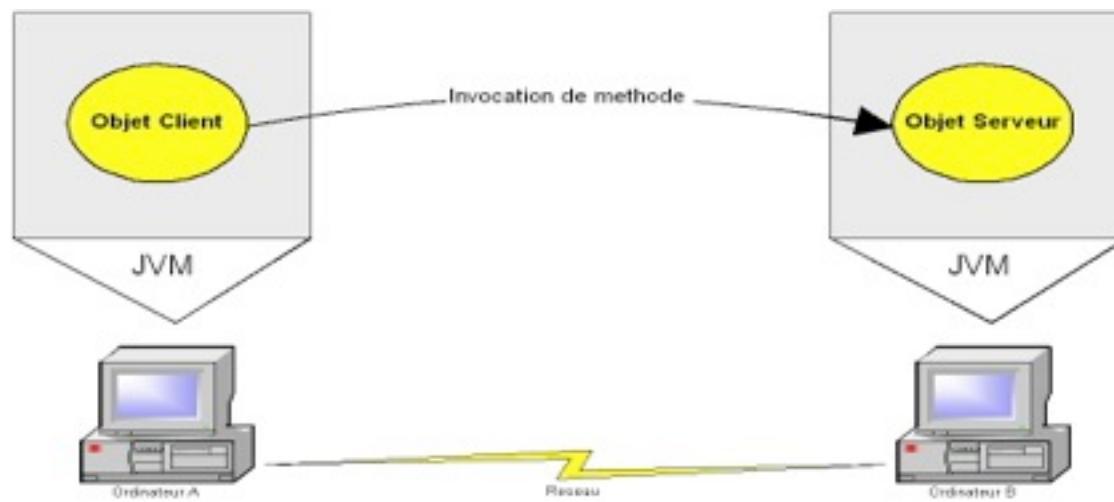
Invocation de méthodes distantes

Mécanisme qui permet à des objets localisés sur des machines distantes de s'échanger des messages (invoquer des méthodes)



Remote Method Invocation

Mécanisme permettant à des objets Java d'invoquer des méthodes sur des objets localisés dans des machines virtuelles différentes (espaces d'adressage distincts), sur le même ordinateur ou sur des ordinateurs distants reliés par un réseau



Remote Method Invocation



- ▶ utilise directement les sockets
- ▶ code ses échanges avec un protocole propriétaire : R.M.P (Remote Method Protocol)
- ▶ RMI est une API (intégré au JDK1.1)
- ▶ RMI est uniquement réservé aux objets Java

Principes (1)

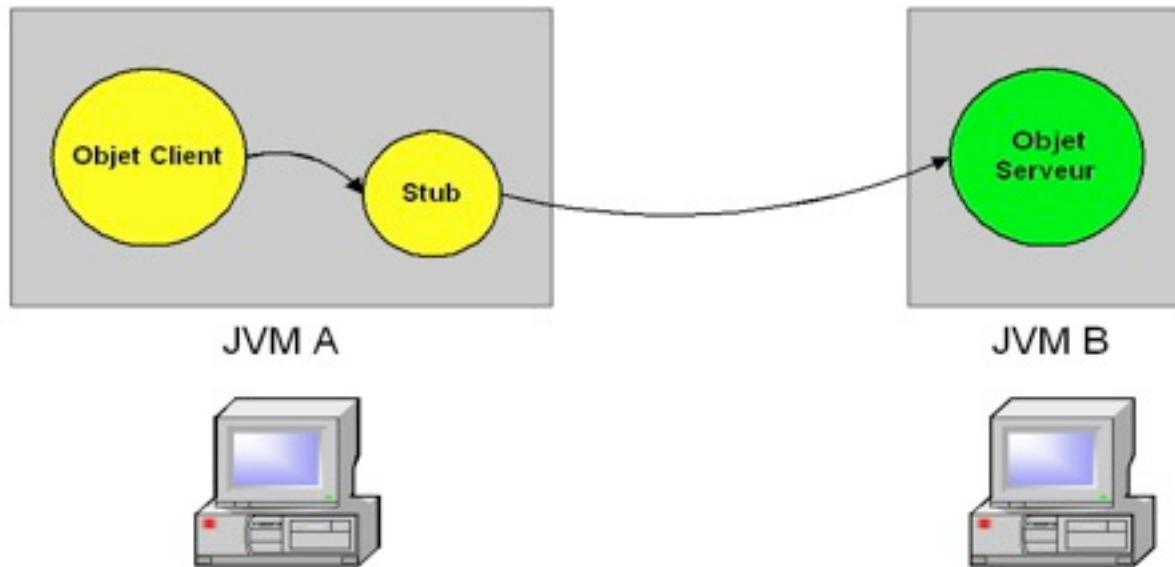


- Un objet distant (objet serveur) : ses méthodes sont invoquées depuis une autre JVM
 - dans un processus différent (même machine)
 - ou dans une machine distante (via réseau)
- Un OD (sur un serveur) est décrit par une interface distante Java
 - déclare les méthodes distantes utilisables par le client

Communication par un intermédiaire

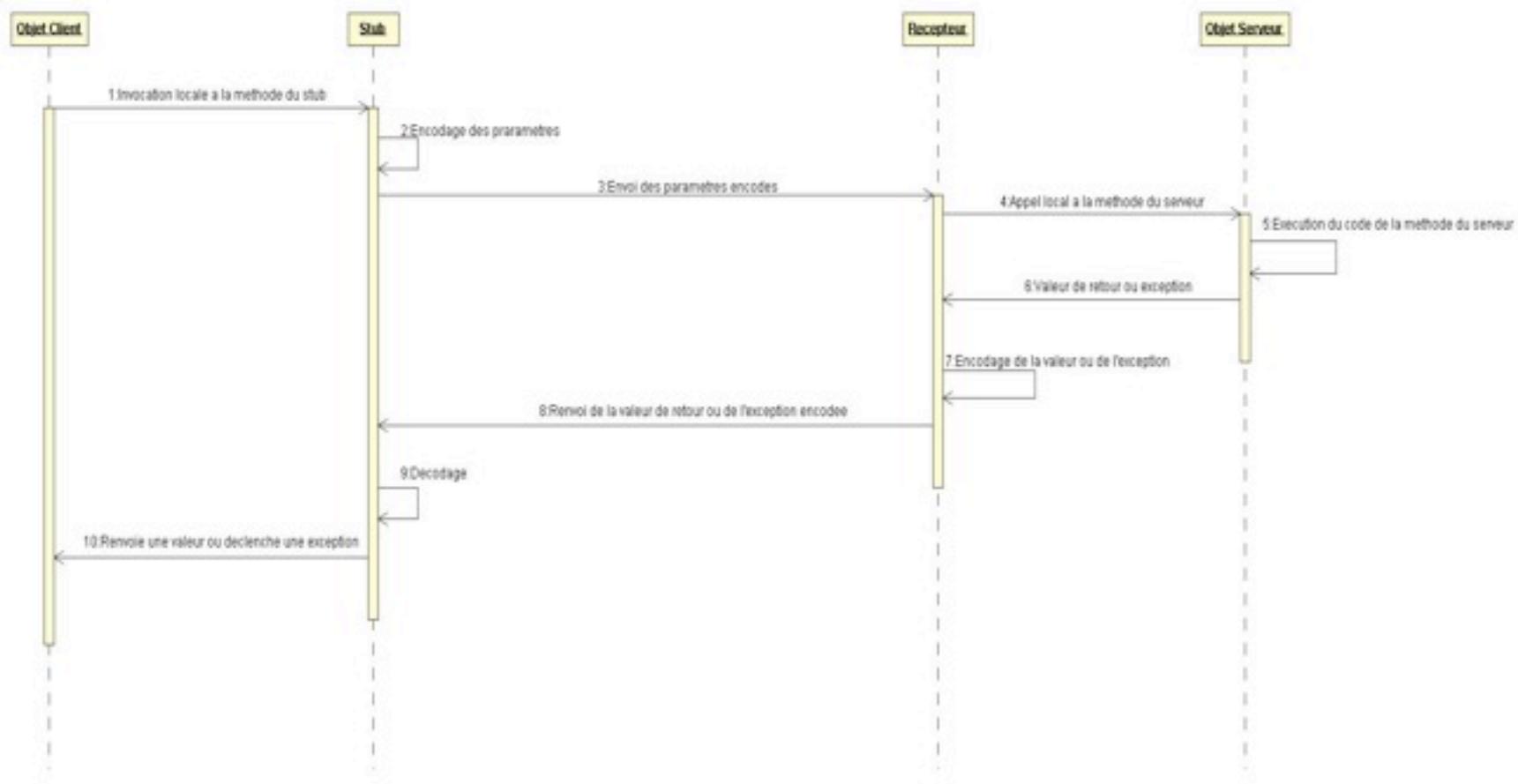
Que se passe-t-il quand le code du client invoque une méthode sur un objet distant ?

- Utilisation d'un objet substitut dans la JVM du client (JVM A) : le stub (« souche ») de l'objet serveur



le Stub est un représentant local d'un objet situé dans un autre espace d'adresses (JVM)

Stubs et encodage des paramètres



Echange Client-Serveur par l'intermédiaire du stub

- Le client appelle une méthode sur un OD
- Il appelle une méthode de l'objet **stub**
- Le **stub** construit un bloc de données avec
 - un identificateur de l'objet distant à utiliser
 - une description de la méthode à appeler
 - les paramètres encodés (*sérialisés*) qui doivent être passés
- Le **stub** envoie ce bloc de données au serveur...
- Lorsque un **objet de réception** reçoit les données
 - Le «*squelette*» effectue les actions suivantes :
 - décode les paramètres encodés (*désérialise*)
 - localise l'objet à appeler
 - invoque la méthode spécifiée
 - capture la valeur de retour ou l'exception renvoyée par l'appel
 - l'encode (*sérialise*) pour transmission vers le client
 - puis le retourne au client

Encodage des paramètres



- Encodage dans un bloc d'octets afin d'avoir une représentation indépendante de la machine et du système d'exploitation
 - Types primitifs et «basiques» (int/Integer, String)
 - encodés en respectant des règles établies
 - codage big-endian pour les entiers...
 - Objets ...processus de sérialisation...

Générateur de stubs



- La commande rmic du jdk rend transparent la gestion du réseau pour le programmeur
 - une référence sur un OD référence en fait son objet souche (stub) local
 - syntaxe = comme un appel local :
 objetDistant.methode()

L'outil rmic



outil livré avec le JDK permet de générer les stubs

```
[user@a01] rmic -v1.2 HelloWorldImpl
```

- génère un fichier *HelloWorldImpl_stub.class* (et un fichier *HelloWorldImpl_Skel.class*)
- La commande rmic doit être effectuée (côté serveur) pour chaque classe d'implémentation d'un OD afin d'en générer les souches.

L'outil rmic



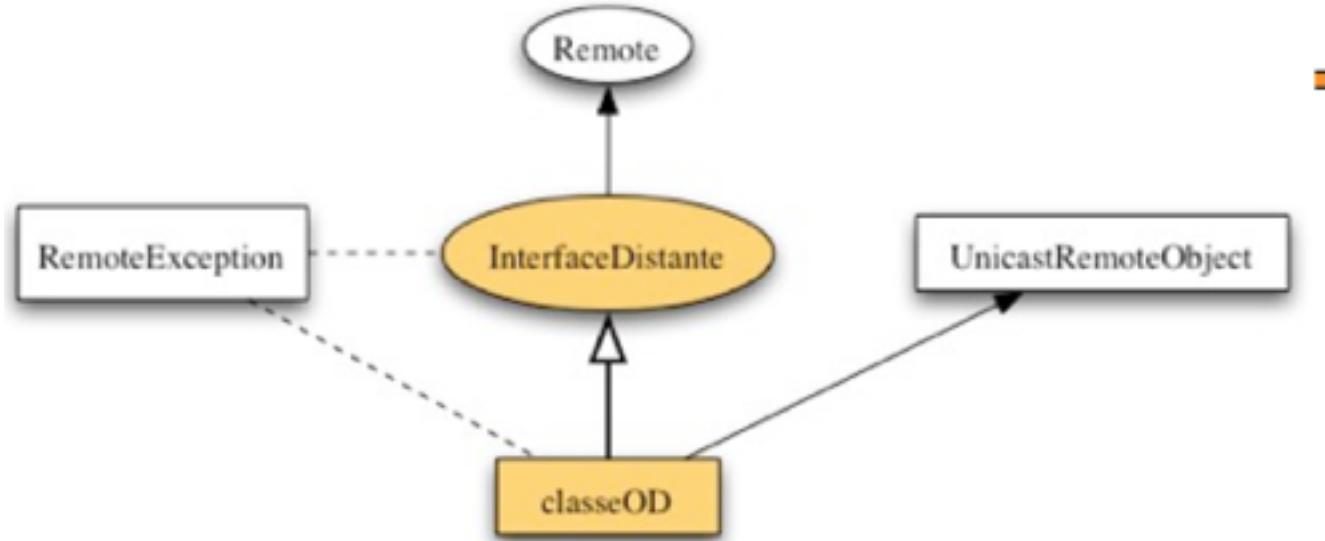
outil livré avec le JDK permet de générer les stubs

```
[user@a01] rmic -v1.2 HelloWorldImpl
```

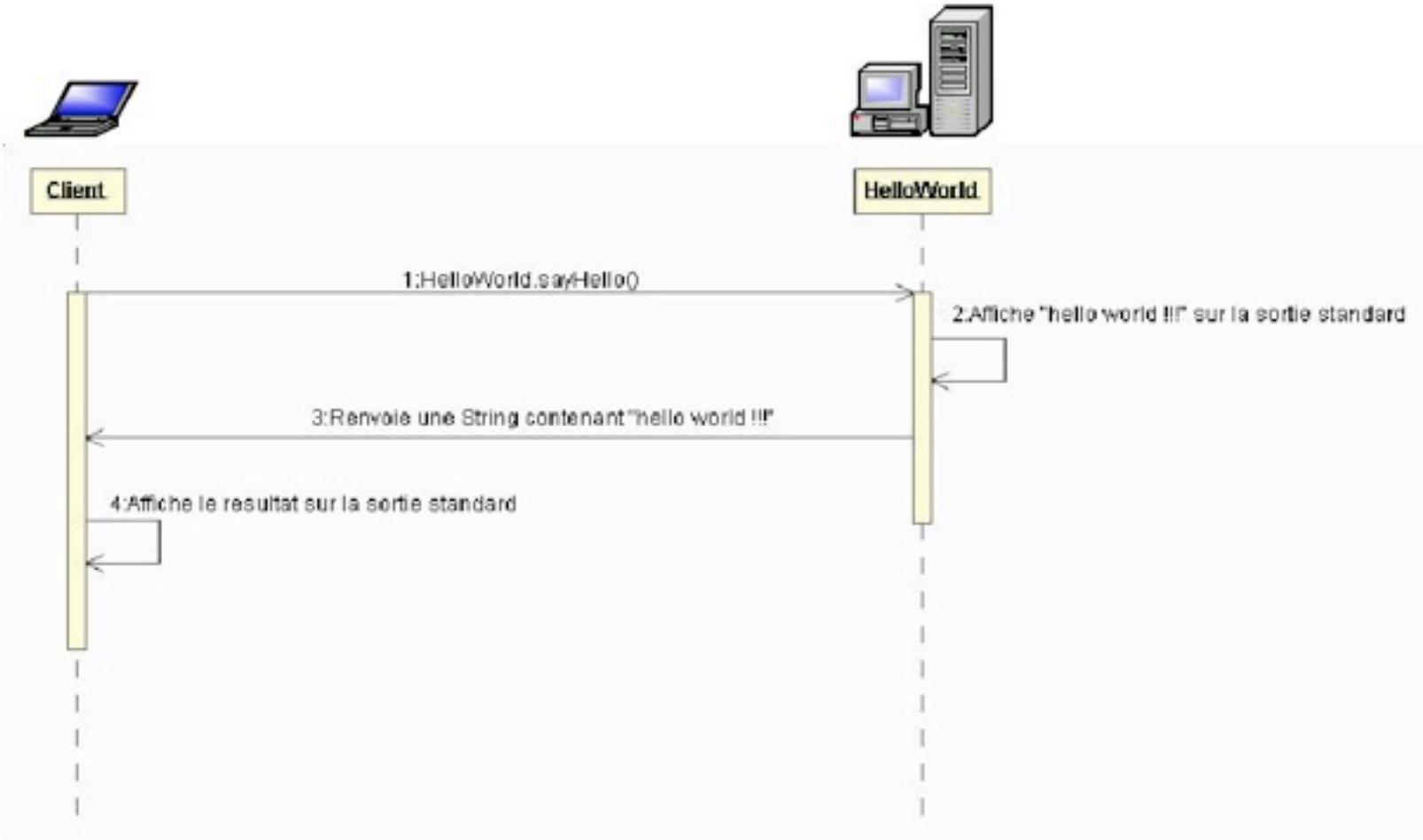
- génère un fichier *HelloWorldImpl_stub.class* (et un fichier *HelloWorldImpl_Skel.class*)
- La commande rmic doit être effectuée (côté serveur) pour chaque classe d'implémentation d'un OD afin d'en générer les souches.

Schéma de réalisation d'une classe

- Toute classe d'OD doit implémenter une interface distante spécifique, précisant quelles méthodes de l'objet sont invocables à distance.
- L'interface distante doit étendre l'interface Remote
- La classe d'OD implémente l'interface distante et étend la classe UnicastRemoteObject
- UnicastRemoteObject est l'équivalent de la classe Object pour les ODs et permet à ses classes filles d'être répertoriées dans le système RMI, donc de recevoir des appels distants.



Un exemple : « Hello World »



Interface = contrat à remplir

- L'interface distante HelloWorld :

```
import java.rmi.*;  
public interface HelloWorld extends Remote {  
    public String sayHello() throws RemoteException; }
```

- *L'exception RemoteException doit être déclarée par toutes les méthodes supportant des appels distants :*
- *Les appels de méthodes distantes sont moins fiables que les appels locaux :*
 - *serveur ou connexion peut être indisponible* • *panne de réseau*
 - *requête sur un OD indisponible*
 - ...

Du côté Serveur

- Implémentation de la classe d'ODs implémentant

- doit implémenter l'interface HelloWorld
- doit étendre la classe RemoteServer du paquetage java.rmi
 - RemoteServer est une classe abstraite
 - UnicastRemoteObject est une classe concrète qui gère la communication avec les stubs
-

```
// Classe d'implémentation du Serveur
public class HelloWorldImp extends UnicastRemoteObject
implements HelloWorld
{
    public String sayHello() throws RemoteException {
        String result = « hello world !!! »;
        System.out.println(« Méthode sayHello invoquée... » +
result); return result;
    }
}
```

Du côté client

- Attention : côté client, toujours référencer un OD en tant qu’instance de l’interface distante (ici HelloWorld) et non de sa classe réelle (ici HelloWorldImpl) !!!

```
HelloWorld hello = ... ;  
    // (nous allons bientôt voir comment obtenir  
    // une première référence sur un stub)
```

```
String result = hello.sayHello();  
System.out.println(result);
```

- L'accès aux champs (attributs) publics de l'OD, n'est pas possible car ils ne sont pas décrits dans l'interface distante.*
- Il faut donc écrire des méthodes accesseurs (getters/ setters) s'il est nécessaire de consulter/modifier les attributs d'un OD ... et déclarer ces accesseurs dans l'interface distante*

Référence sur un objet



On sait implanter un serveur d 'un côté, et appeler ses méthodes de l 'autre

MAIS

Comment obtient-on une référence vers un stub de notre objet serveur ???

Localisation des objets de serveur



- ▶ *Pour appeler une méthode sur un autre objet serveur* --> il faut avoir un référence sur le Stub
- ▶ *Pour avoir un référence sur le Stub* --> il faut localiser l'objet qui est de l'autre côté du canal de communication
- ▶ *Pour localiser l'objet* --> on a besoin d'un Service de Nommage

Les Services de Nommage



- ▶ Enregistrement des références d'ODs dans un annuaire (service de nommage) afin que des programmes distants puissent les récupérer.
- ▶ Obtention d'une première référence sur un OD : « bootstrap » à l'aide de l'annuaire.
- ▶

Les Services de Nommage



- ▶ Implémentation d'un service de nommage
- ▶ Fourni en standard avec RMI
- ▶ Permet d'enregistrer des références sur des objets de serveur afin que des clients les récupèrent
- ▶ On associe la référence de l'objet à une clé unique (chaîne de caractères)
- ▶ Le client effectue une recherche par la clé, et le service de nommage lui renvoie la référence distante (le stub) de l'objet enregistré pour cette clé

Le RMI Registry



- Programme exécutable fourni pour toutes les plates formes
- S'exécute sur un port (1099 par défaut) sur la machine serveur
- Pour des raisons de sécurité, seuls les objets résidant sur la même machine sont autorisés à lier/délier des références
- Un service de nommage est lui-même localisé à l'aide d'une URL

La classe Naming



- du package java.rmi
 - permet de manipuler le RMIRRegistry
 - supporte des méthodes statiques permettant de
 - Lier des références d'objets serveur
 - Naming.bind(...) et Naming.rebind(...)
 - Délier des références d'objets serveur
 - Naming.unbind(...)
 - Lister le contenu du Naming
 - Naming.list(...)
 - Obtenir une référence vers un objet distant
 - Naming.lookup(...)

Enregistrement d 'une référence



- L 'objet serveur HelloWorld (coté serveur bien entendu...)
 - On a créé l'objet serveur et on a une variable qui le référence

```
HelloWorld hello = new HelloWorldImpl();
```
 - On va enregistrer l'objet dans le RMIRegistry

```
Naming.rebind("HelloWorld",hello);
```
 - L'objet est désormais accessible par les clients

Obtention d'une référence côté client



- sur l'objet serveur HelloWorld
 - On déclare une variable de type HelloWorld et on effectue une recherche dans l'annuaire

```
HelloWorld hello = (HelloWorld)Naming.lookup("rmi://server/  
hello");
```

- On indique quelle est l'adresse de la machine sur laquelle s'exécute le RMIRegistry ainsi que la clé
- La valeur renournée doit être transtypée (castée) vers son type réel

Processus de développement d'une application RMI

On veut invoquer la méthode echo() d'un objet de serveur distant depuis un programme Java client

- Nous allons devoir coder
 - L'objet distant
 - Echo : l'interface décrivant l'objet distant (OD)
 - EchoImpl : l'implémentation de l'OD
 - Le serveur
 - EchoAppliServer : une application serveur RMI
 - Le client
 - EchoClient : l'application cliente utilisant l'OD
 - Et définir les permissions de sécurité et autres emplacements de classes...

Processus de développement d'une application RMI

- 1) définir une interface Java pour un OD
- 2) créer et compiler une classe implémentant cette interface
- 3) créer et compiler une application serveur RMI
- 4) créer les classes Stub et Skeleton (**rmic**)
- 5) démarrer **rmiregistry** et lancer l'application serveur RMI
- 6) créer, compiler et lancer un programme client accédant à des OD du serveur

1) Définir l'interface pour la classe

- Rappel : les classes placées à distance sont spécifiées par des interfaces qui doivent dériver de **java.rmi.Remote** et dont les méthodes lèvent une **java.rmi.RemoteException**

```
import java.rmi.*;  
  
public interface Echo extends Remote  
{  
    public String echo(String str) throws RemoteException;  
}
```

2) Définir l'implémentation de l'OD

```
import java.rmi.*;
import java.rmi.server.*;

public class EchoImpl
extends UnicastRemoteObject implements Echo
{
    public EchoImpl() throws RemoteException {super(); }
    public String echo(String str) throws RemoteException
    {
        return "Echo : " + str;
    }
}
```

Définir une application serveur

- ▶ instancie un objet de type HelloWorld
- ▶ pour rendre l'objet disponible, il faut l'enregistrer dans le «RMIServer» (service de nommage) par la méthode statique :
 Naming.rebind("echo", od);
- ▶ puis objet mis en attente des invocations jusqu'à ce que le serveur soit arrêté

```
public class EchoAppliServer
{
    public static void main(String args[])
    {
        try
        {
            // Création de l'OD
            EchoImpl od = new EchoImpl();
            // Enregistrement de l'OD dans RMI
            Naming.rebind("echo", od);
        }
        catch (Exception e) { ... }
    }
}
```

Définir une application serveur (suite)

- Par la suite, cet objet distant sera accessible par les autres machines en indiquant son nom et la machine sur laquelle est exécuté ce serveur par l'url :
 - `String url = "rmi://nomServeurRMI:port/nomOD";`
- Cet "url RMI" sera utilisé par les clients pour interroger le serveur grâce à l'appel :
 - `Naming.lookup(url);`

Compiler les classes



```
> javac Echo.java  
> javac EchoImpl.java  
> javac EchoAppliServer.java
```

- Ces commandes créent les fichiers .class correspondants.

4) Créer les amorces



- Il s'agit d'invoquer le compilateur d'amorces **rmic** sur la classe compilée de l'OD :
 - > **rmic EchoImpl**
- 2 classes sont alors créées représentant la souche (ou Stub) et le Skeleton :
 - **EchoImpl_Stub.class**
 - **EchoImpl_Skel.class**

5) Lancer rmiregistry et l'application



- Il faut d'abord lancer le **rmiregistry** puis le serveur :
 - > **rmiregistry&**
 - > java EchoAppliServer
- Ce qui rend maintenant le serveur disponible pour de futurs clients ...

6) Définir l'application cliente utilisant

- ▶ obtenir une référence sur l'objet de serveur Echo,
- ▶ invoquer la méthode echo(),
- ▶ puis afficher le résultat de l'invocation sur la sortie standard

```
import java.rmi.*;  
import java.rmi.registry.*;  
  
public class EchoClient  
{  
    public static void main(String args[])  
    {  
        // Recherche de l'OD  
        String url = "rmi://" + args[0] + "/echo";  
        Echo od = (Echo)Naming.lookup(url);  
        System.out.println(od.echo(args[1]));  
    }  
}
```

6) Définir l'application cliente utilisant



- Il suffit ensuite de lancer le programme
 - > java EchoClient
 - Recherche de l'objet serveur...
 - Invocation de la méthode echo...
 - Affichage du résultat :
hello world !!!
- Remarque :
 - ce code manipule l'OD comme s'il était local.
 - le client recherche l'objet distant en interrogeant le service « d'annuaire » RMI par :
 - **Naming.lookup(url)** ou url est un « url RMI » de la forme:
 - **String url="rmi://nomServeurRMI:port/nomOD"**

Que doit connaître le client ?



- Lorsqu'un objet serveur est passé à un programme, soit comme paramètre soit comme valeur de retour, ce programme doit être capable de travailler avec le stub associé
- Le programme client doit connaître la **classe** du stub, (ex. EchoImpl_Stub.class)
 - copier les classes sur le système de fichiers local du client (CLASSPATH)...