# Objets distants exemple en Java

## Remote Method Invocation


À travailler en TPs


Concepts généraux

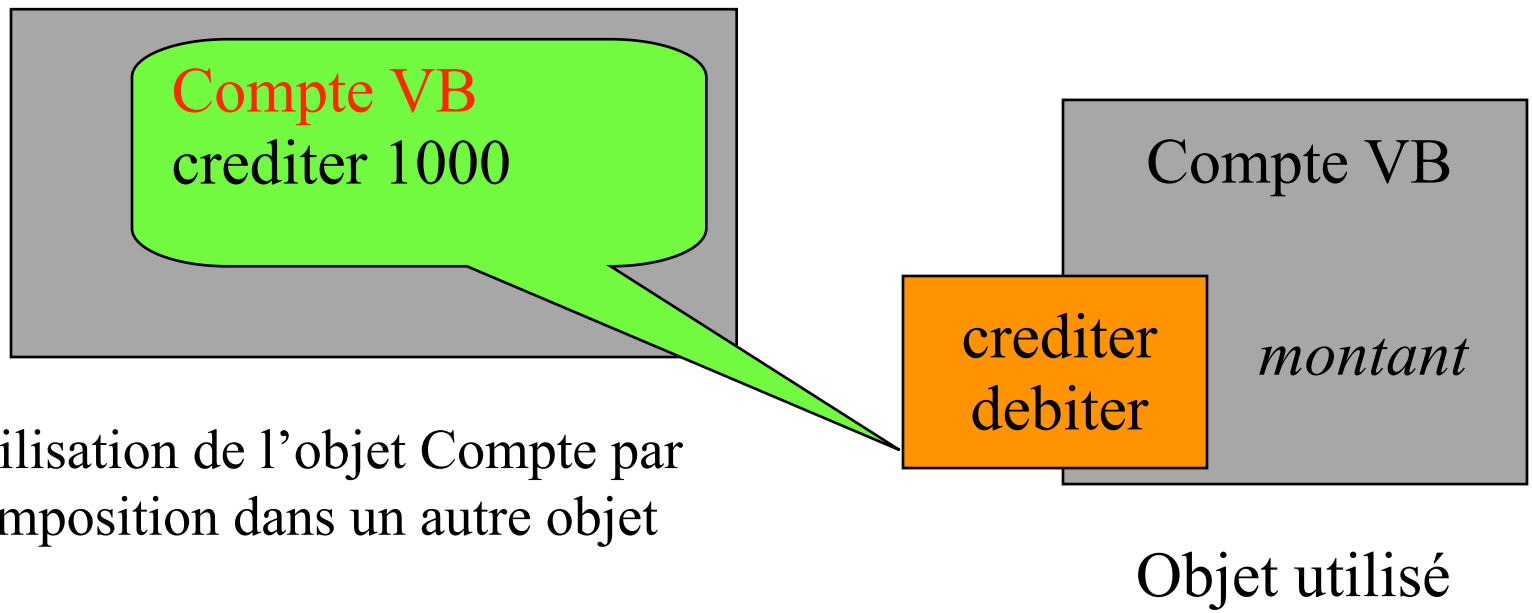

Mise en œuvre Java

*Document établi depuis celui de A.M. Dery et collabs.*

# Points forts de la programmation par Objets

- Encapsulation, modularité
  - Application = collection d'objets interagissant
- *Mécanismes d'abstraction*
  - *Séparation entre interface et implémentation*
- *Représentation et types de données*
  - *Hiérarchie de classes, surcharge*

# Interface publique



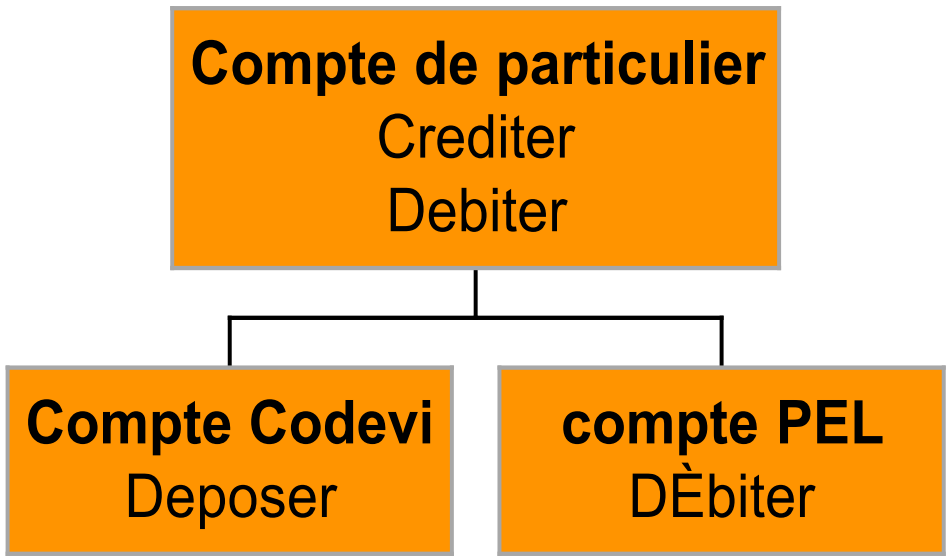

Utilisation de l'objet Compte par composition dans un autre objet

Objet utilisé

**Interface** = partie visible de l'objet
**Implémentation** = partie privée inaccessible depuis d'autres objets

Modifier l'implémentation des classes *sans altérer leur utilisation*, car on doit respecter le contrat entre l'objet et le monde extérieur

# Classes et héritage



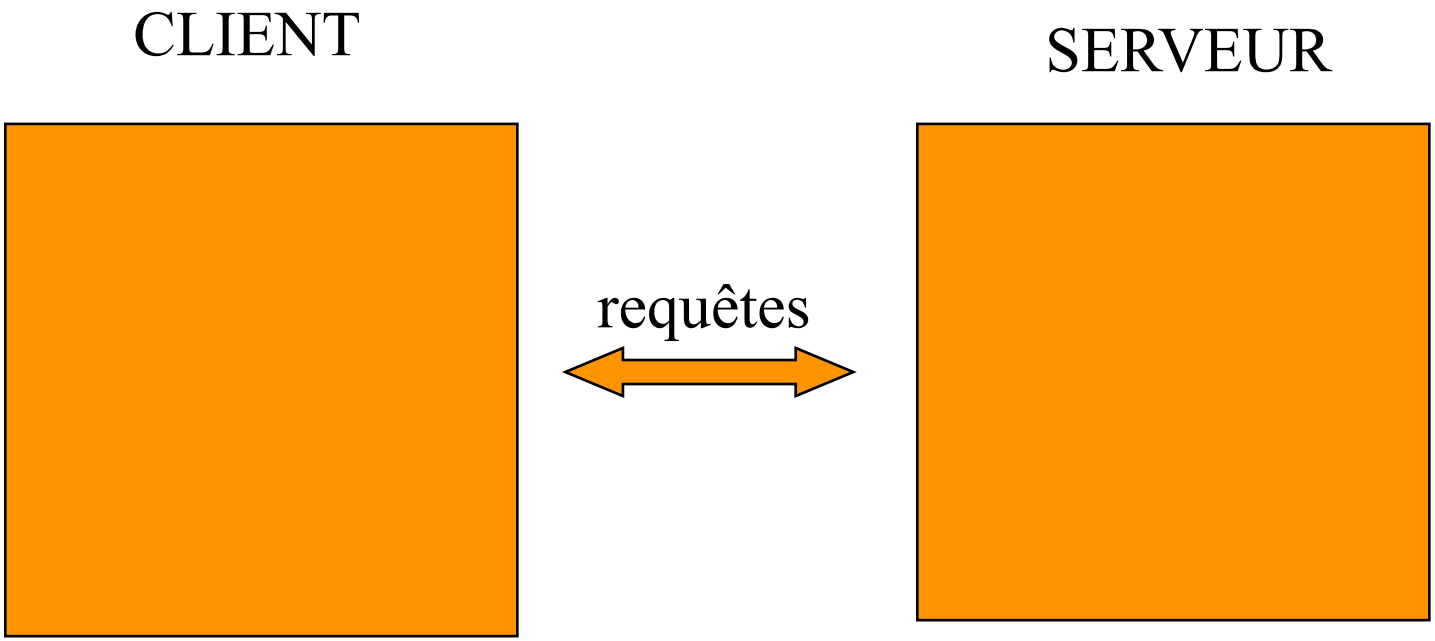

Ajout de nouveaux sous-types par création de classes

Ajout et surcharge des méthodes

# Communications Client / Serveur

- Programmes (fonctionnant sur des machines différentes) communiquant au travers du réseau.
- Un programme Serveur offre des services (publie des fonctionnalités)
- Un programme Client utilise les fonctionnalités par envoi de requêtes à un programme serveur (qui prend en charge l'implémentation des fonctionnalités)

# Infrastructure Client / Serveur



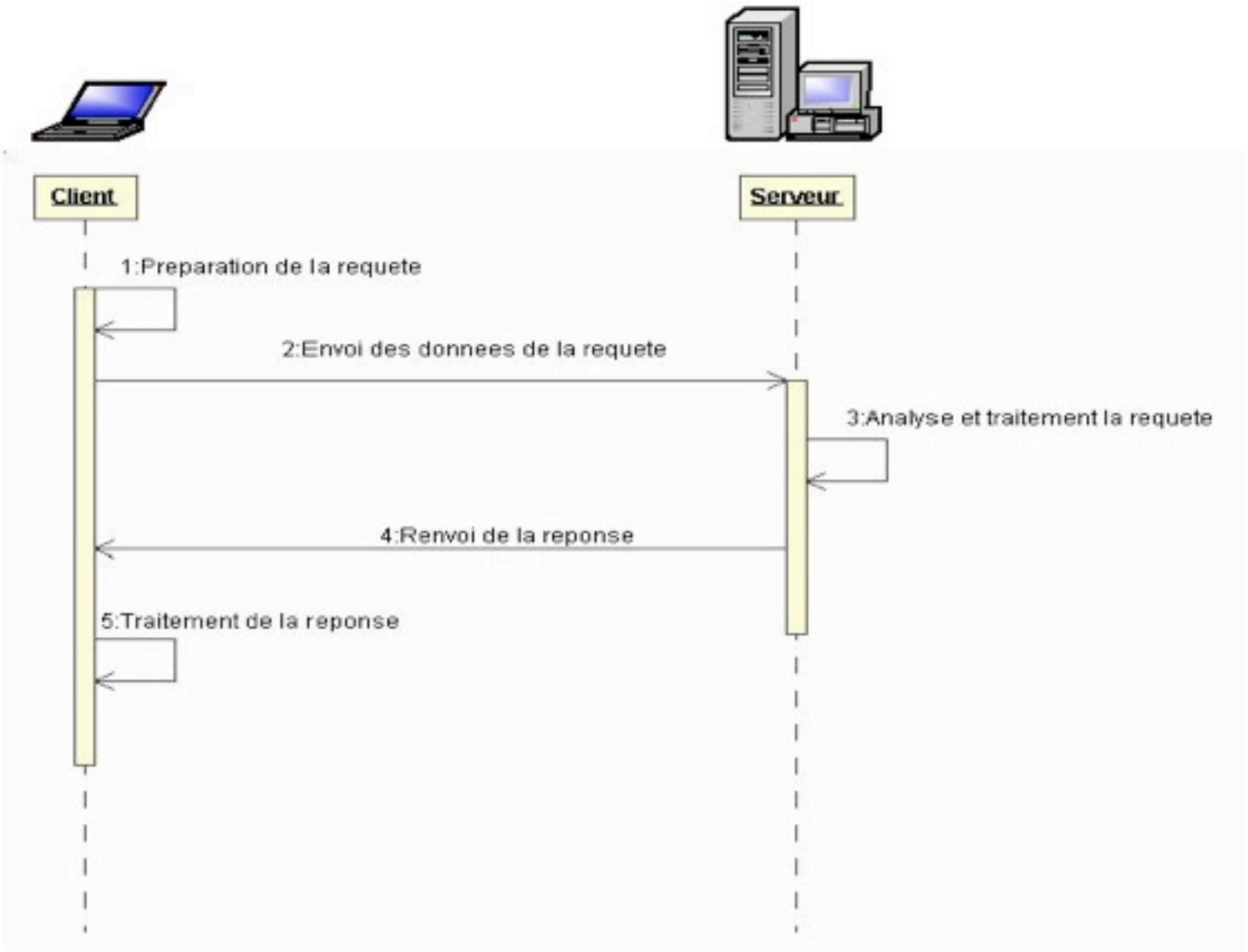

Ensemble des requêtes = ***protocole d'application*** du service.

*Exemple* :

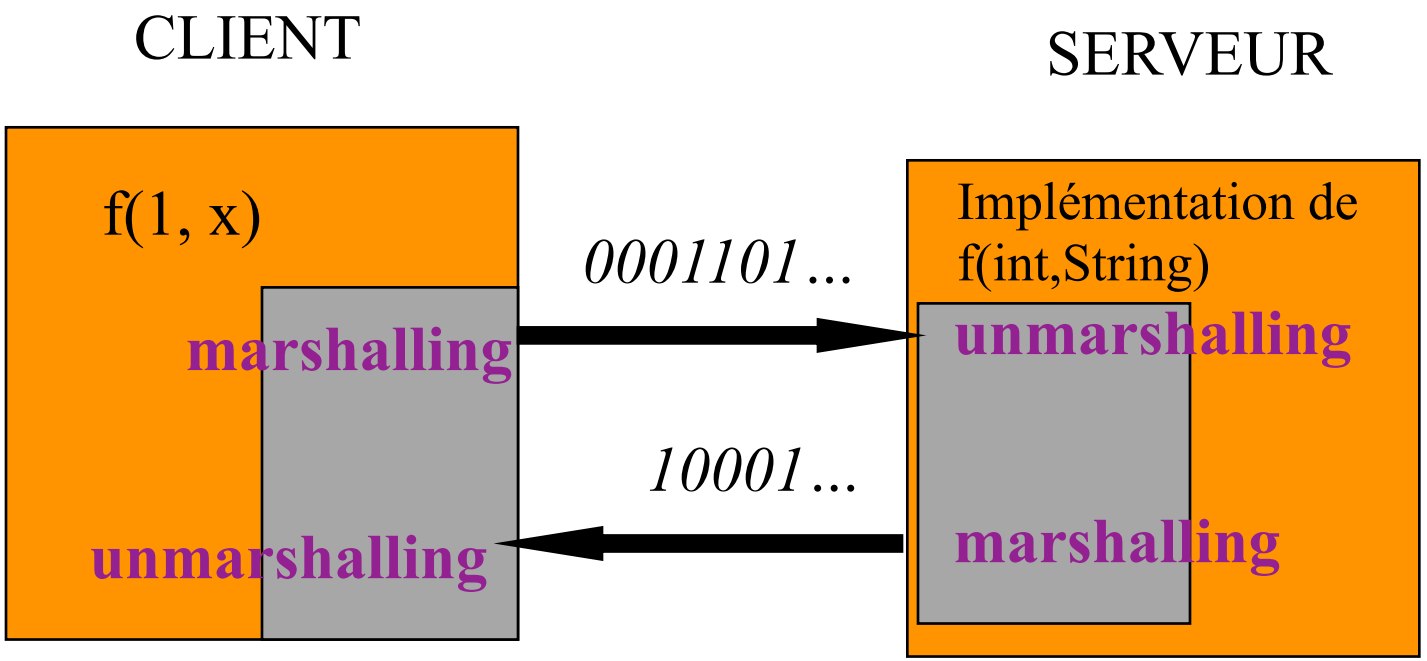
1 - authentification

2 - opération bancaire

# Client / Serveur



# Appel de procédure à distance




***Marshalling*** : Préparer le format et le contenu de la trame réseau
**Unmarshalling** : Reconstruire l'information échangée à partir de la trame réseau

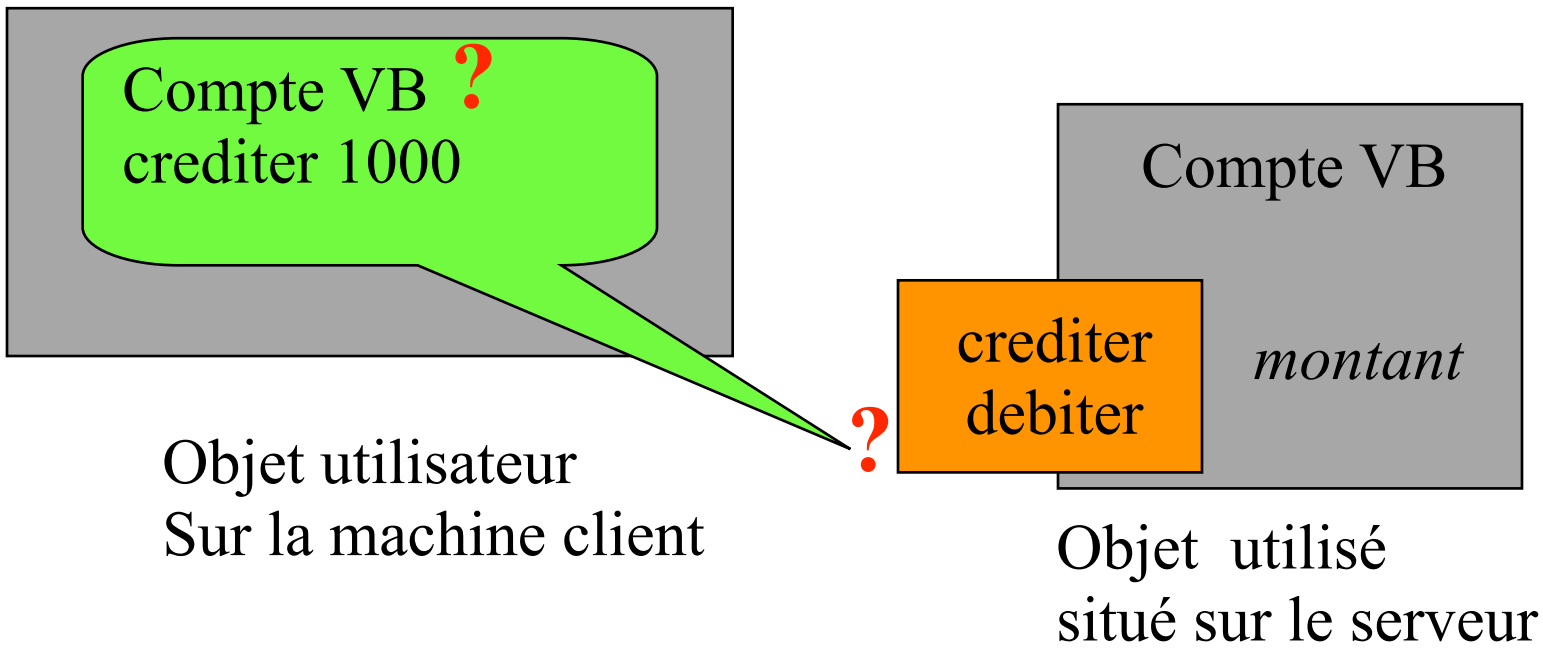
# Objets Distants

• 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.

• Comme les objets normaux, les objets distants sont passés par référence (l'objet est donc modifiable).

• Indépendamment de l'endroit d'utilisation de la référence, l'invocation de méthode se produit dans l'objet d'origine, qui réside toujours sur la machine d'origine.

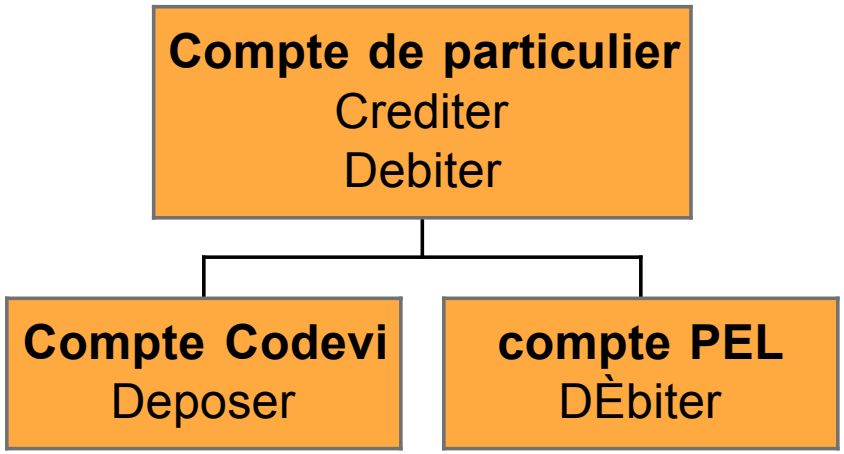
# Utilisation d'objets distants




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

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

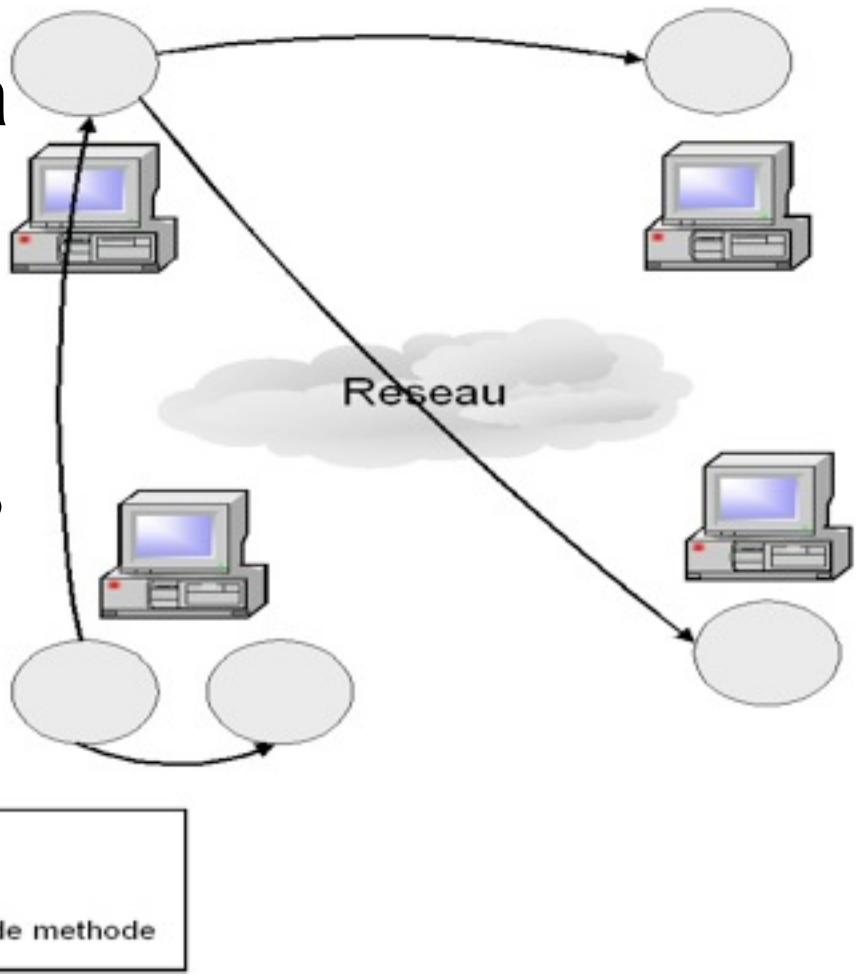
# Objets Distants et héritage




Spécialisation d'un serveur : changement de versions

# Invocation de méthodes à distance

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



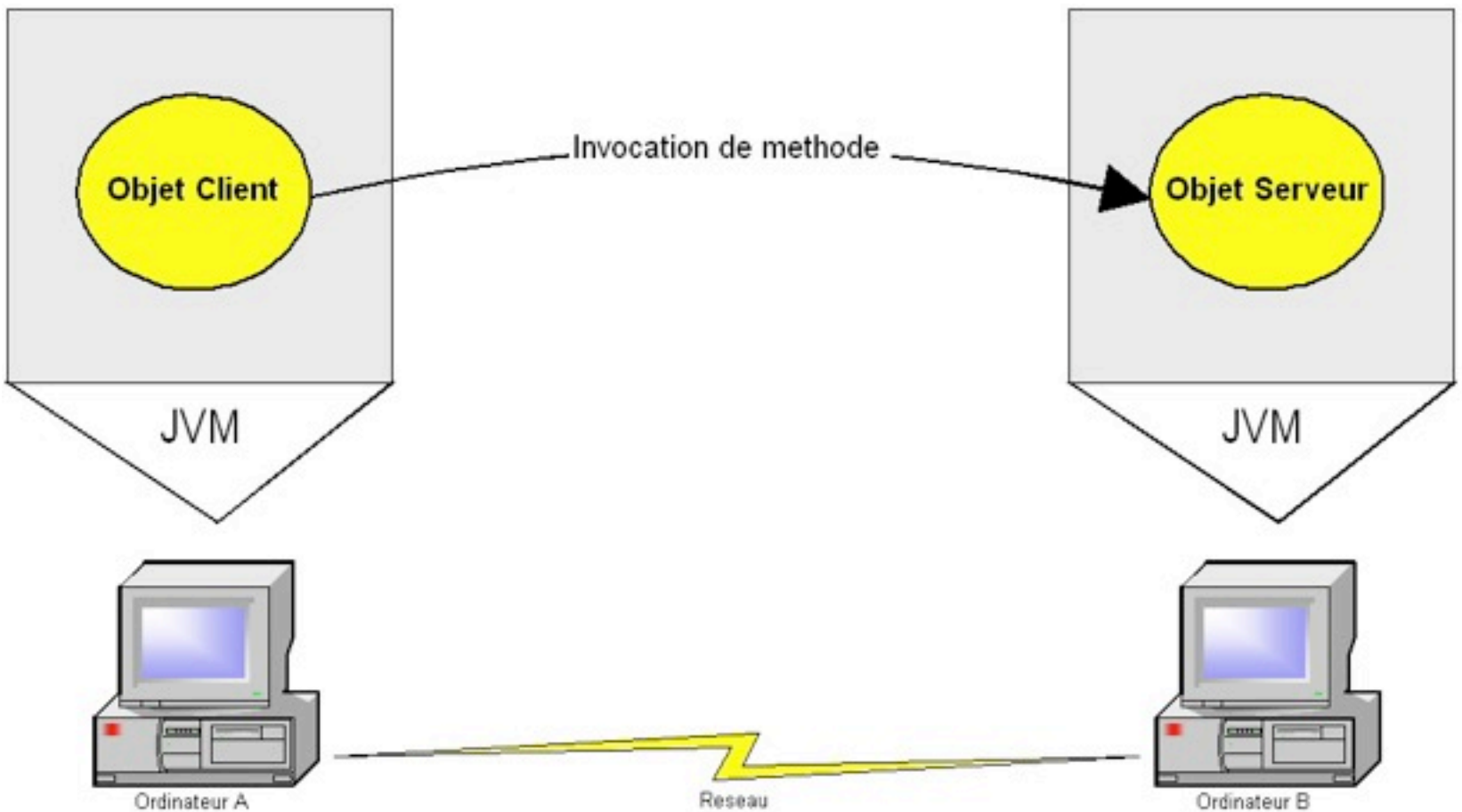
# Invocation de méthodes distantes

- Semble simple en théorie...
- ... un peu plus complexe en réalité !!!

# RMI

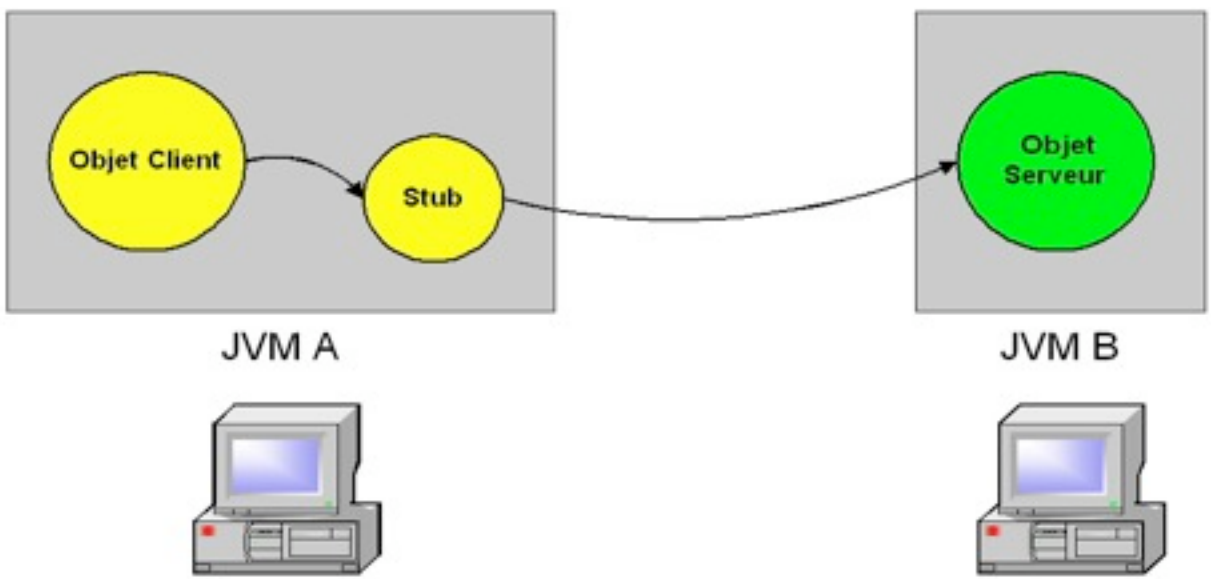
- On va illustrer la façon de faire Java :
  - Java *Remote Method Invocation* est la solution à notre problème
  - *RMI* permet à des objets Java d'invoquer des méthodes sur des objets localisés dans des JVM différentes, et même distantes sur le réseau, et ceci de façon quasi transparente !!!

# Invocation de méthodes distantes

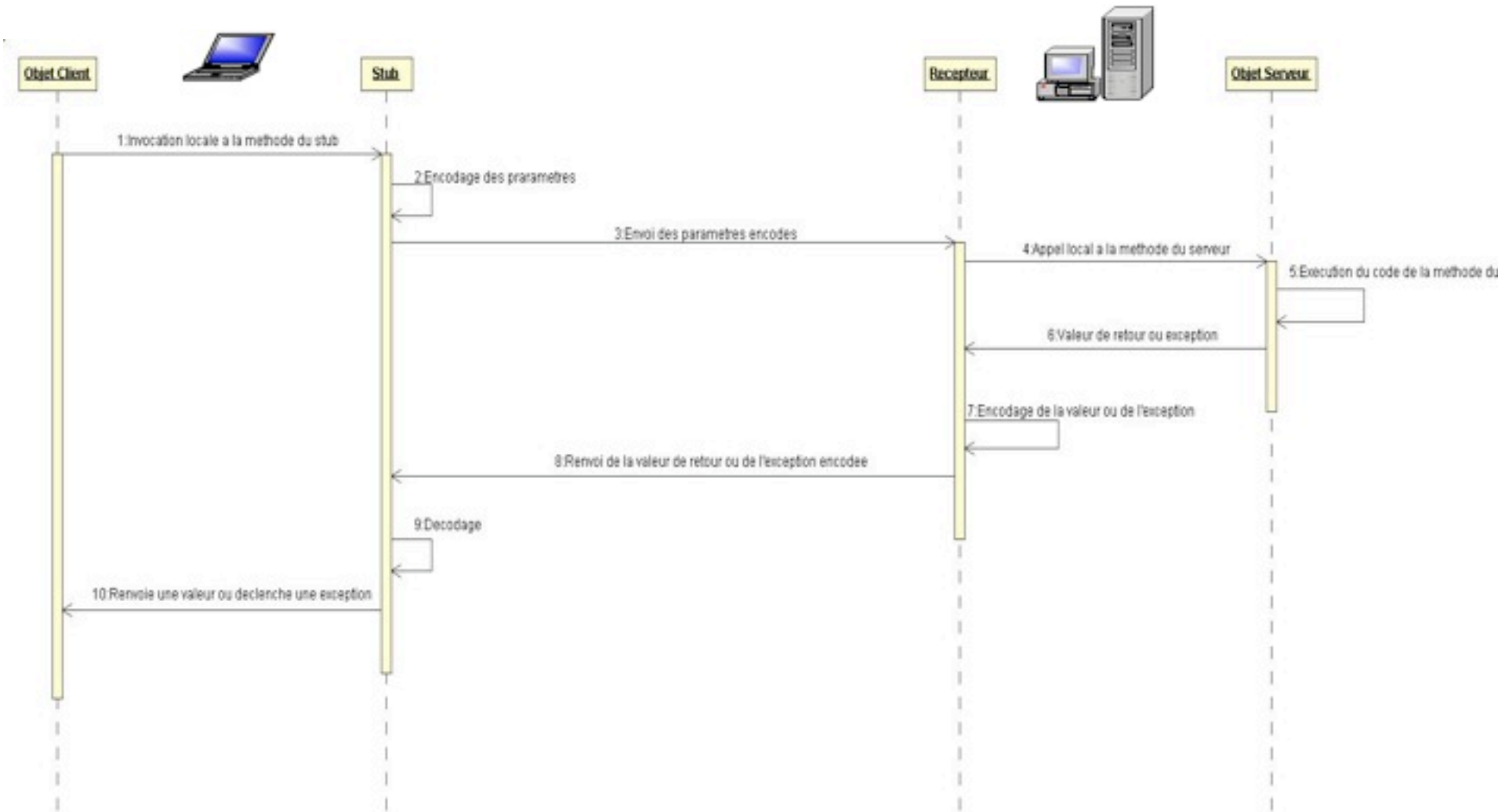


# Communication par un intermédiaire

- 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



# L’intermédiaire stub



# 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

# Patron de conception Procuration (Proxy)

- Le stub est un exemple du patron ***Procuration*** ( on dit aussi « mandataire », en anglais « proxy ») :
  - L'utilisation d'une **procuration** est indiquée quand on a besoin d'une référence à un objet qui soit plus créative et plus sophistiquée qu'un simple pointeur.
  - Notamment ici, le stub est une ***procuration à distance,*** c-a-d fournissant un représentant local d'un objet situé dans un autre espace d'adresses (JVM)

# Patron de conception Procuration (Proxy)

- Le stub est un exemple du patron ***Procuration*** ( on dit aussi « mandataire », en anglais « proxy ») :

– Utlisé quand un objet ne désire pas être dérangé trop souvent ou doit être protégé de l'extérieur (flic méchant / flic gentil).

*Sens du terme proxy sur un réseau*

– Une autre application est la ***procuration virtuelle***, c-a-d créant des objets « lourds » à la demande; ils ne sont réellement instanciés dans leur intégralité qu'au moment (en cas) d'un accès aux fonctionnalités lourdes des objets.

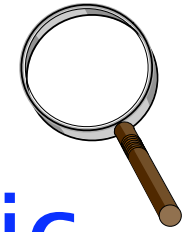
*Exemple = chargement d'images dans un document de plusieurs pages dont une seule est visualisée à la fois*

# 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…

# Génération de stubs par rmic

- 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()`
- Eclipse facilite la création des stubs



# L'outil rmic

- outil livré avec le JDK permet de générer les stubs

  [user@a01] rmic -v1.2 HelloWorldImpl

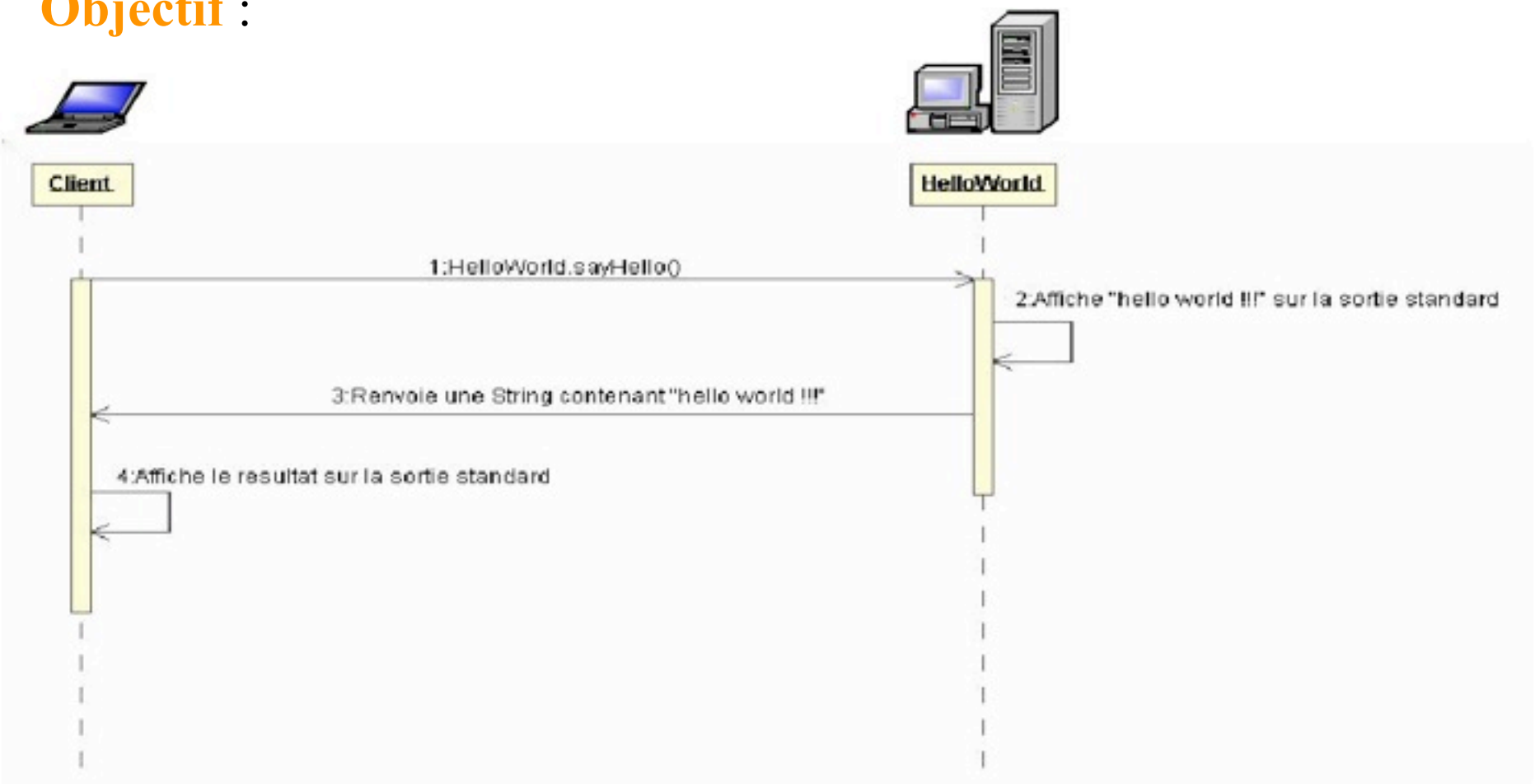
- génère un fichier HelloWorldImpl_stub.class

  (et un fichier HelloWorldImpl_Skel.class)

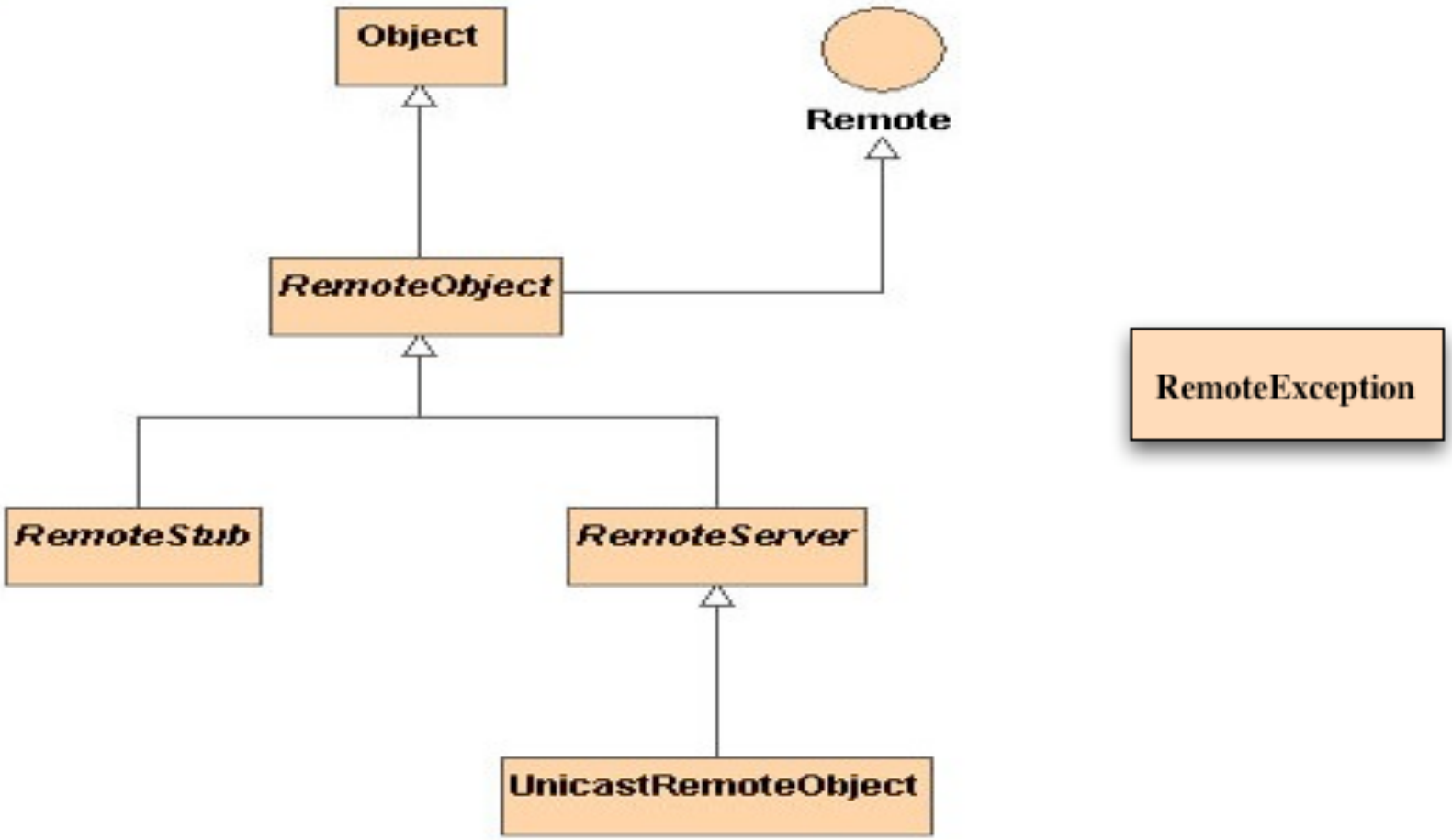
  La commande *rmic* doit être effectuée (côté serveur) pour chaque classe d'implémentation d'un OD afin d'en générer les souches.

# Un exemple : « Hello World »

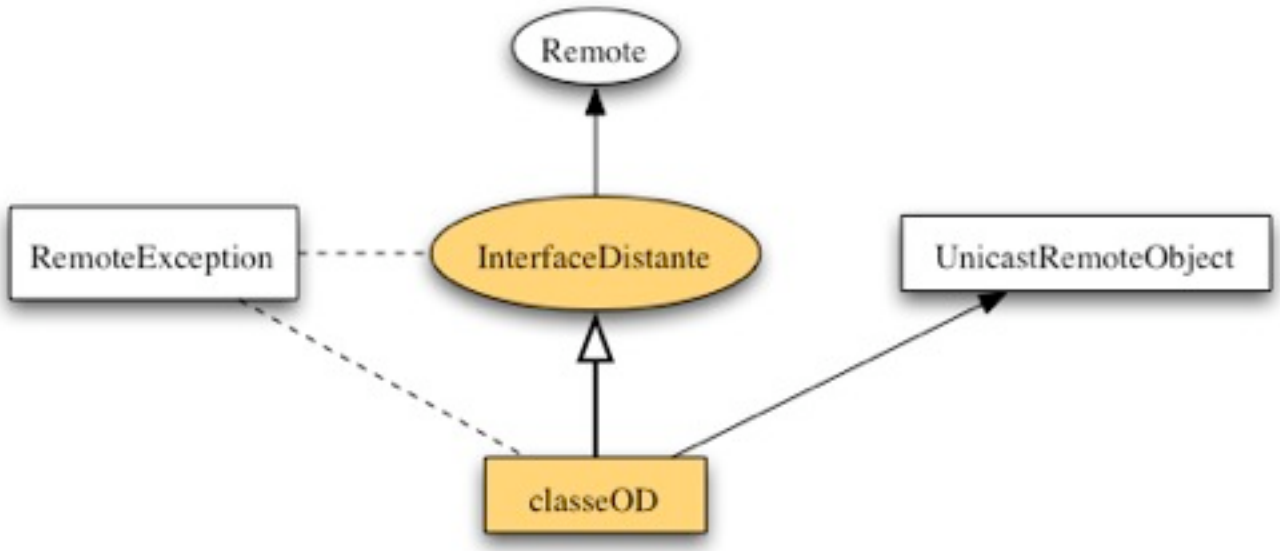
**Objectif** :



# Interfaces et classes prédéfinies de Java



# Schéma de réalisation d'une classe d'ODs



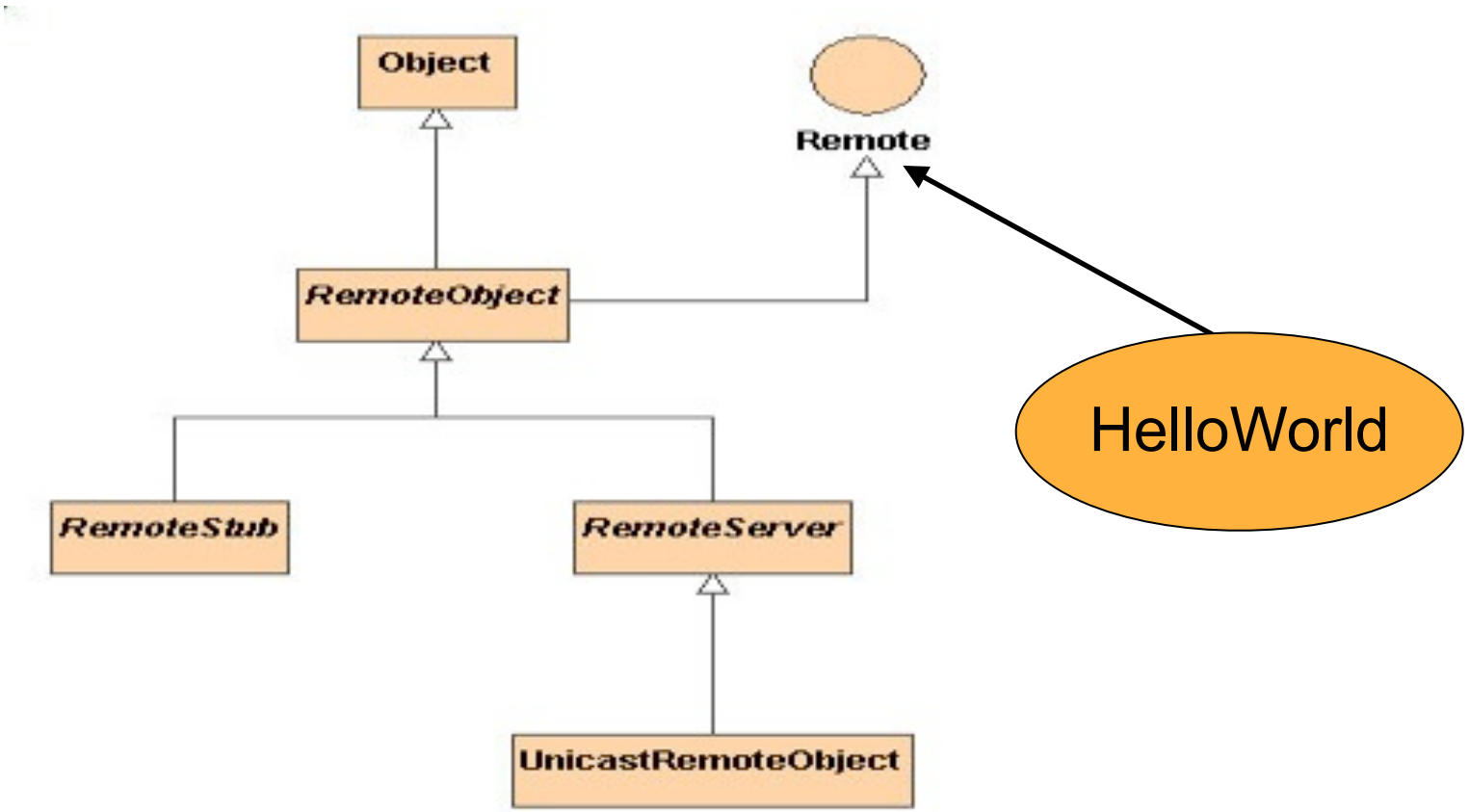

- Toute classe d'OD doit implémenter une interface distante spécifique, précisant quelles méthodes de l'objet sont invocables à distance.
- L'interface distante doit étendre l'interface Remote
- La classe d'OD *implémente* l'interface distante et étend la classe UnicastRemoteObject
- UnicastRemoteObject 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.

# Interface = contrat à remplir

- L'interface distante HelloWorld :

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

# A l'exécution, dans la JVM



# Les exceptions

- 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

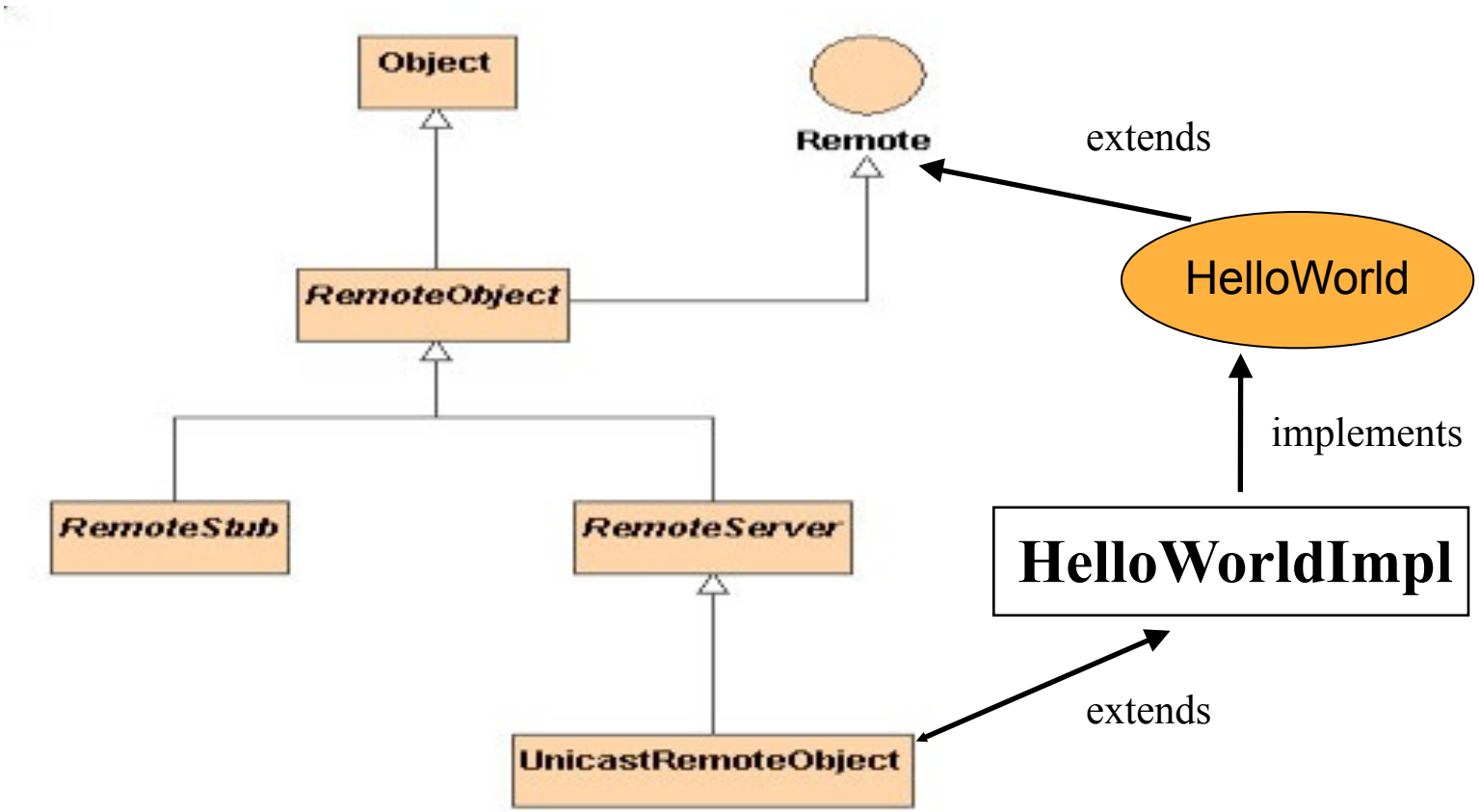
- Implémentation de la classe d'ODs implémentant l'interface `HelloWorld` :

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

# Classe d'implémentation

- 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 gére la communication avec les stubs

# A l'exécution, objet de la classe d'implémentation



# 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) !!!

Comme la souche réalise (*implements*) elle aussi cette interface distante, ils sont interchangeables en ce qui concerne l'invocation de méthodes :

```
HelloWorld hello = ...      ;
   // (nous allons bientôt voir comment obtenir
   // une première référence sur un stub)
String result = hello.sayHello();
System.out.println(result);
```

# Du côté client

- 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 du côté client

MAIS

- Côté client, comment obtient-on une référence vers le stub d’un OD ?

# Localisation des objets du serveur

- On pourrait appeler une méthode sur un autre objet serveur qui renvoie une référence sur le stub...
- Mais comment localiser ce premier objet qui est de l'autre côté du canal de communication ?

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.

# Exemple : le RMIRegistry

- Implémentation d'un service de nommage
- Fourni en standard avec RMI

- Permet d'enregistrer des références sur des objets gérés par le serveur afin que des clients les récupèrent

- On associe la référence de l'objet à une clé unique (chaîne de caractères) (comme un dictionnaire)
- La clef est communiquée aux clients par un autre biais

- Un 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 RMIRegistry

- Programme exécutable fourni pour toutes les plateformes
- S'exécute sur un port (1099 par défaut) sur la machine serveur
- Pour des raisons de sûreté, seuls les objets résidant sur la même machine sont autorisés à lier/délier des références, c-a-d remplir ce registre
- mais ce registre peut bien-sûr être consulté à distance (par les clients)
- 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 RMIRegistry
  - contient 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 la référence d'un Objet Distant
      - Naming.lookup(...)

# Enregistrement d'une référence

- Pour l'objet serveur HelloWorld *(côté serveur bien entendu…).*
  - Dans le code on a créé l'objet serveur et on a une variable qui le référence :

```
HelloWorld hello = new HelloWorldImpl();
```

  - On enregistre ensuite l'OD dans le RMIRegistry au moyen d'une clé :

```
Naming.rebind("UnHelloWorld",hello);
```

On peut indiquer une clef plus compliquer en précisant un serveur et un numéro de port, par ex "//localhost:1099/UnHelloWorld"

- L'objet est désormais accessible par les clients

# Obtention d'une référence côté client

- sur l'objet serveur HelloWorld
  - On déclare une instance de l'interface distante HelloWorld et on effectue une recherche dans l'annuaire de nommage sous la forme d'une URL utlisant le protocole rmi :

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

  - On indique quelle est **l'adresse de la machine** sur laquelle s'exécute le RMIRegistry (avec éventuellement un numéro de port, par ex machine.ici.fr:1099) ainsi que la clé associée à l'OD.
  - La valeur retournée doit être transtypée (castée) vers le type attendu car le registre renvoie un Object.

# Remarques

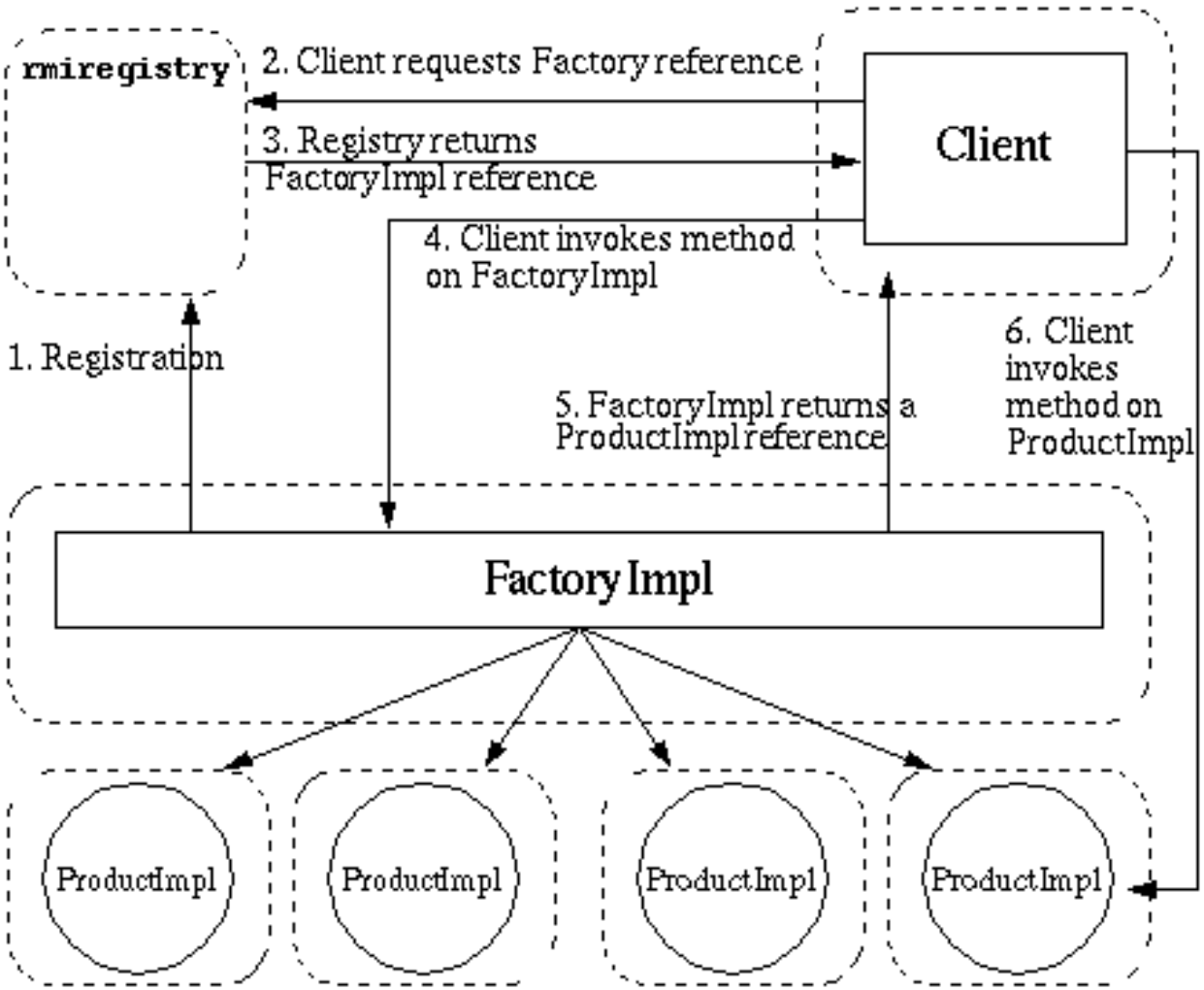
- Le service de nommage n'a pas pour fonction de référencer tous les objets du serveur
  - il devient vite complexe de gérer l'unicité des clés
  - il s'agit d'un service assez lent

- La règle de bonne utilisation du *Naming* est de lier des objets qui font office de point d'entrée, et qui permettent de manipuler les autres ODs : **patron de conception** *Factory* (Fabrication).
- Tire parti du fait qu'un appel à une méthode sur un OD peut renvoyer une référence sur un autre OD, ainsi obtenu sans passer par le registre rmi !

# Patron de conception *Fabrication* pour l'accès aux ODs

• Ce patron est utile quand on a besoin qu'un objet contrôle la création et/ou l'accès à d'autres objets.

• En utilisant ce patron avec RMI, on réduit le nombre d'objets qu'il est nécessaire de déclarer dans le registre RMI (limite les risques de collisions de clés + accélère les accès).



# Conception, implémentation, mise en place et exécution de l'exemple

- Rappel
  - On veut invoquer depuis un client en Java la méthode `sayHello()` d'un objet de type `HelloWorld` siué sur un serveur distant
- Nous allons devoir coder
  - L'objet distant (OD)
  - Le serveur
  - Un client
  - Et définir la liaison client-serveur (les permissions de sécurité et autres emplacements de classes...)

# Processus de déploiement

Côté Serveur

- 1) **Définir le protocole d'application** - définir une interface Java pour un OD
- 2) **Implémenter le service** - créer et compiler une classe implémentant cette interface
- 3) **Offrir le service** - créer et compiler une **application serveur** RMI
- 4) **Gérer la communication** - créer les classes Stub (**`rmic`**)
- 5) **Lancer le serveur de nommage** - démarrer **`rmiregistry`**
- 6) lancer l'application serveur RMI

Côté Client

- 7) Définir une **application client** qui appelle les services offerts par le serveur en se basant sur le protocole d'application (interface des ODs)
- 8) créer, compiler et lancer le code accédant à des ODs du serveur

# Hello World : L'objet distant

- Une interface et une classe d'implémentation
- stubs générés automatiquement par la commande rmic
- *toutes les classes nécessaires pour manipuler les ODs côté client doivent être déployées sur la machine cliente et accessibles au chargeur de classes (c-a-d, dans le* CLASSPATH*) :*
  - l'interface HelloWorld (HelloWorld.class)
  - le stub HelloWorldImpl_Stub.class généré par rmic pour ce type d'ODs.

# Hello World : le serveur

- instancie un objet de type HelloWorld et l'attache au service de nommage
- puis l'OD est mis en attente des invocations jusqu'à ce que le serveur soit arrêté :

```
import java.rmi.*;
import java.rmi.server.*;
public class HelloWorldServer {
        public static void main(String[] args) {
                try {
                    System.out.println("Création de l'objet serveur...");
                    HelloWorld hello = new HelloWorldImpl();
                    System.out.println("Référencement dans le RMIRegistry...");
                    Naming.rebind("rmi://server/unHelloWorld",hello);
                    System.out.println("Attente d'invocations - CTRL-C pour stopper");
                } catch(Exception e) { e.printStackTrace(); }
        }
}
```

# Serveur (suite)

1. On compile d'abord les interfaces et les classes
2. On lance ensuite le RMIRegistry
   - *Attention* : La base de registres RMI doit connaître les interfaces et les stubs des objets qu'elle enregistre (CLASSPATH) !!!

```
> rmiregistry &  (start rmiregistry sous Windows)
```

3. Puis on lance le serveur :

```
> java HelloWorldServer

Création de l'objet serveur...
Référencement dans le RMIRegistry...
Attente d'invocations - CTRL-C pour stopper
```

# Hello World : client

Ensuite on met en place les clients :

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

```
import java.rmi.*;
        public class HelloWorldClient {
        public static void main(String[] args) {
            try {
                System.out.println("Recherche de l'objet serveur...");
                HelloWorld hello = (HelloWorld)Naming.lookup("rmi://server/
unHelloWorld");                    System.out.println("Invocation de la méthode sayHello...");
                String result = hello.sayHello();
                System.out.println("Affichage du résultat :");
                System.out.println(result);
                System.exit(0);
            } catch(Exception e) { e.printStackTrace(); }
```

# Client (suite)

- Après compilation de la classe client on lance le programme :

```
> java HelloWorldClient

Recherche de l'objet serveur...
Invocation de la méthode sayHello...
Affichage du résultat :
hello world !!!
```

- Au niveau du serveur, le message...

```
Méthode sayHello invoquée... hello world !!!
```

...s'affichera dans la console

C’est la PAUSE !

# Passage de paramètres

- On sera souvent amené à passer des paramètres aux méthodes distantes...
- Les méthodes distantes peuvent retourner une valeur ou lever une exception...
- On a deux types de paramètres
  - Les objets locaux au client
  - Les objets distants au client

# Passage de paramètres: ATTENTION

- Certains objets sont copiés (les objets locaux), d'autres non (les objets distants)
- Les objets distants, non copiés, résident sur le serveur
- Les objets locaux passés en paramètre doivent être sérialisables afin d'être copiés (transmis), et ils doivent être indépendants de la plate-forme
    - Les objets qui ne sont pas sérialisables lèveront des exceptions
    - Attention aux objets sérialisables qui utilisent des ressources locales !!!

# Que doit connaître le client ?

- Lorsqu'un objet distant 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: HelloWorldImpl_Stub)

# Que doit connaître le client ?

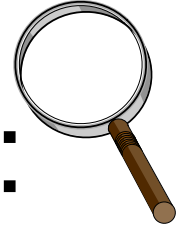
- les classes des paramètres, des valeurs de retour et des exceptions doivent aussi être connues...
  - Une méthode distante est déclarée avec un type de valeur de retour...
  - ...mais il se peut que l'objet réellement renvoyé soit une sous-classe du type déclaré

# Que doit connaître le client ?

- Le client doit disposer des classes de stub pour les classes des ODs retournés…
  - copier les classes sur le système de fichiers local du client (CLASSPATH)...
  - ...cependant, si le serveur est mis à jour et que de nouvelles classes apparaissent, il devient vite pénible de mettre à jour le(s) client(s).
  - C'est pourquoi les clients RMI peuvent charger automatiquement des classes de stub depuis un autre emplacement :

    (il s'agit du même type de mécanisme pour les applets qui fonctionnent dans un navigateur)

# Hello World : chargement dynamique



Séparation des classes :

- **Serveur (fichiers nécessaires à l'*exécution du serveur*)**
  - HelloWorldServer.class : serveur proposant des ODs
  - HelloWorldImpl.class : classe d'ODs
  - HelloWorld.class : interface d'ODs
  - HelloWorldImpl_Stub.class : accès aux ODs (nécessaire pour le registre)
- **Download** (**fichiers de classes à charger dans le programme client**)
  - HelloWorldImpl_Stub.class : stubs pour communiquer avec les ODs.
- **Client** (**fichiers nécessaires au *démarrage* du client**)
  - HelloWorld.class : interface des ODs
  - HelloWorldClient.class : classe cliente

# Les points à approfondir

- Comment cela fonctionne au niveau du réseau.
- Le fonctionnement des applications réparties
  - Chargement dynamique : mise en œuvre
  - Passage de paramètres
  - Notion d'annuaire
  - Autres types de communications …

# Comment cela fonctionne au niveau du réseau

- Rôle du stub :
  - Identification de la machine qui abrite le serveur par le client
  - Identification de l'application serveur sur la machine serveur
  - Canal de communication entre le serveur et le client
  - Construction de la trame réseau (*marshalling*)
  - Echange du protocole d'application

# Passage de paramètres

- On sera souvent amené a passer des paramètres aux méthodes distantes...
- Les méthodes distantes peuvent retourner une valeur ou lever une exception...

- On a deux types de paramètres
  - Les objets locaux
  - Les objets distants

# Chargement dynamique de classes

- Utilise la capacité qu'à tout programme Java de charger dynamiquement du code d'une URL dans sa JVM.
  - En conséquence, un utilisateur peut exécuter un programme, comme une applet, qui n'a jamais été installé sur son disque.
- RMI offre cette capacité à tous les programmes Java (même non-web)
  - Entre autres, un client peut donc charger des classes de type stub pour lancer des appels de méthodes sur des ODs s'exécutant sur des machines distantes.

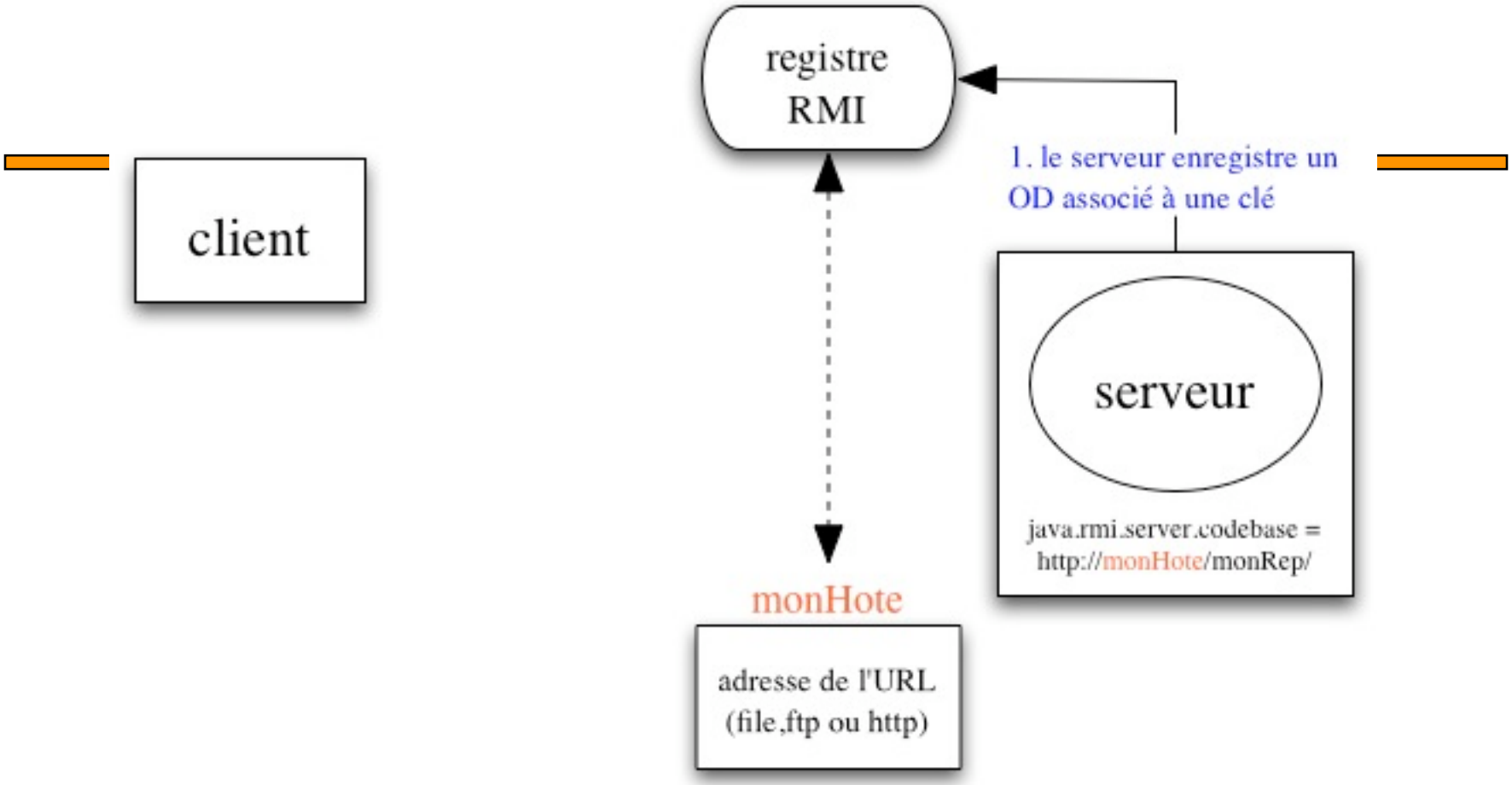
# Chargement dynamique de classes

- Notion de codebase :
  - C'est un endroit où trouver des classes à exécuter dans une JVM. Cet endroit est en général situé sur une autre machine.
  - Notre CLASSPATH peut être vu comme un codebase local
  - Un codebase peut être déclaré du côté serveur ou du côté client pour des utilisations différentes.
  - Il s'agit en général d'une machine tiers (c-a-d, différente du serveur et du client).

# Chargement dynamique de classes

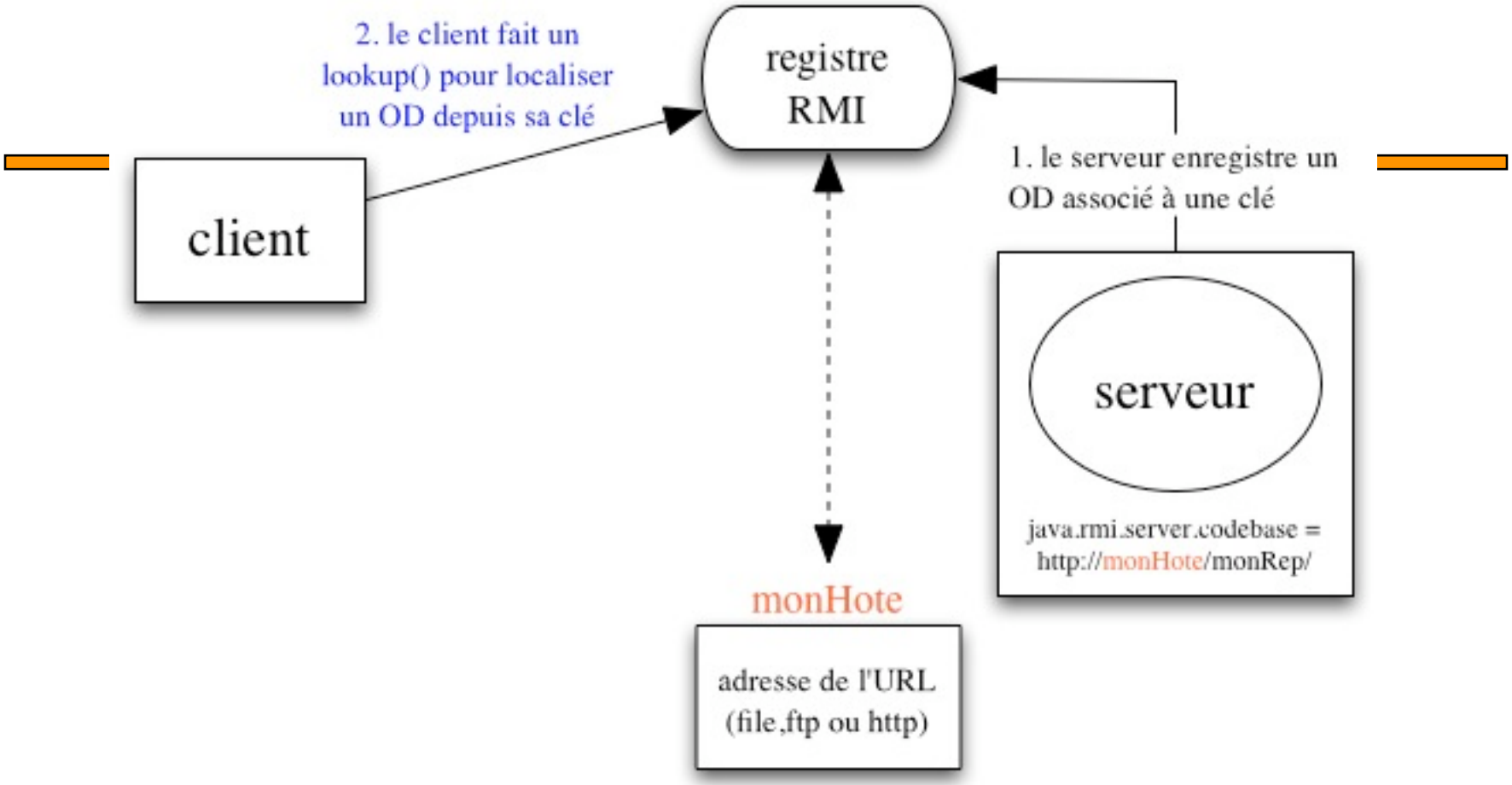
- Le codebase a le format d'une URL :
  - file://monRep/ indique un endroit sur le disque de la même machine que celle se référant à ce codebase; ou sur le même site (en cas de montage NFS).
  - En général les classes chargées par codebase le sont à distance, c-a-d on utilise plutôt les protocoles http ou ftp, par exemple
    - http://monHote.unSite.fr/monRep/
- Syntaxe du codebase : on peut indiquer
  - un répertoire (ne pas oublier le **/** final)
  - une archive jar
  - un ensemble d'endroits (séparés par des espaces)

# Chargement dynamique de classes



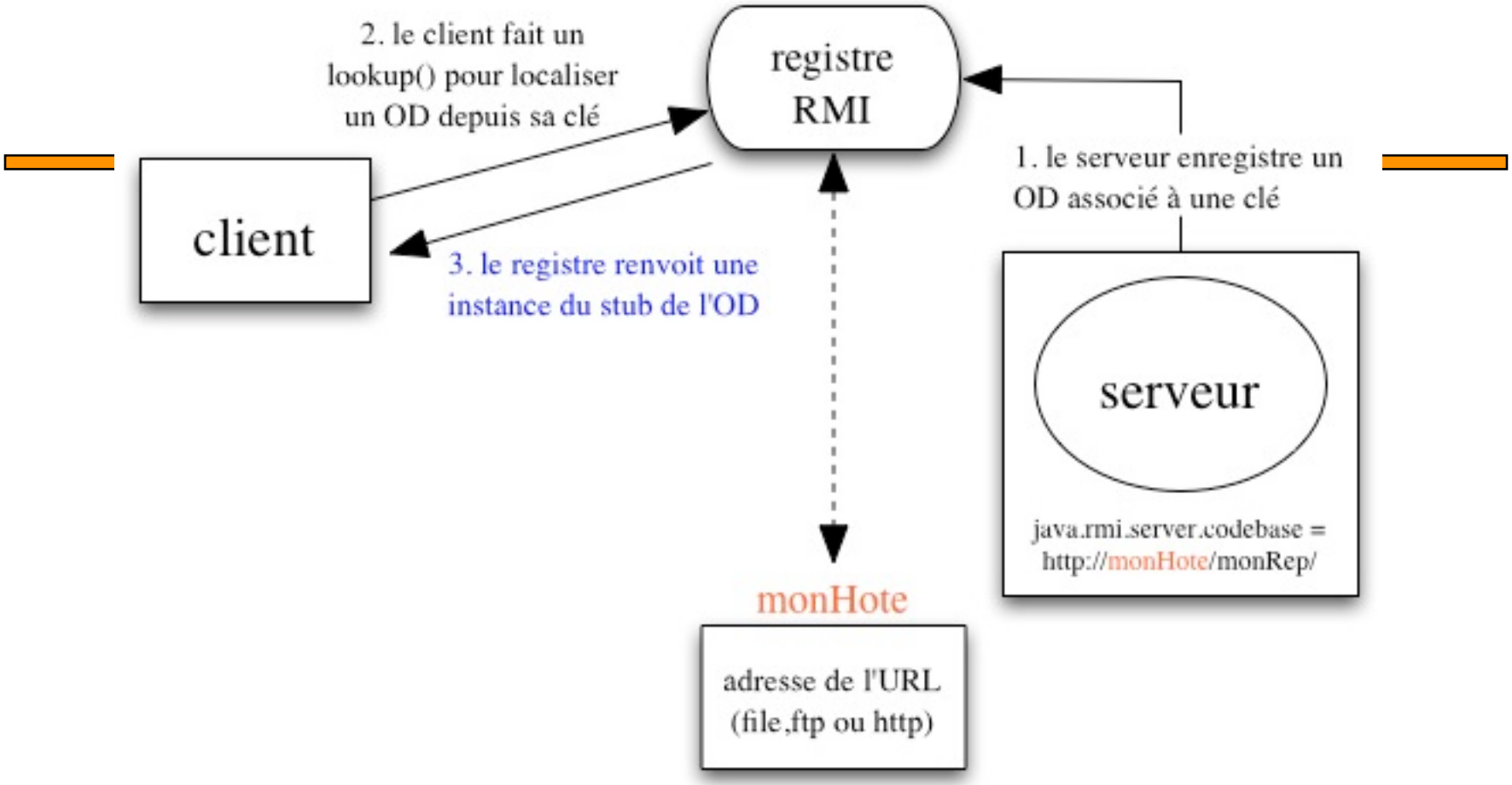

1. Le codebase d'ODs est spécifié par le serveur d'ODs dans la propriété java.rmi.server.codebase; puis le serveur enregistre un/des ODs dans le registre RMI en les associant à des clés. Le codebase spécifié est indiqué avec la référence à l'OD dans le registre RMI.

# Chargement dynamique de classes



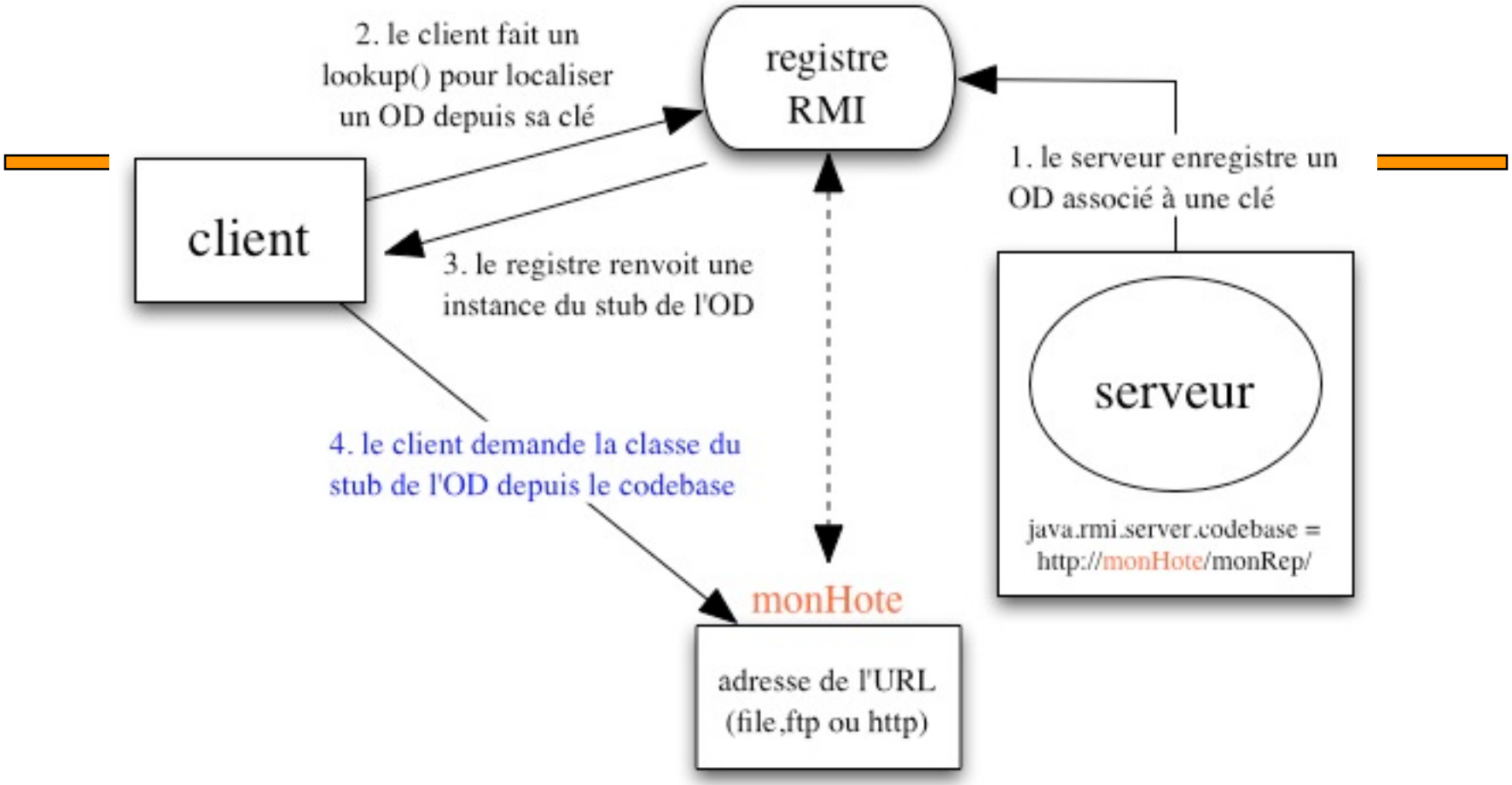

2. Le client demande une référence à un OD dont il précise le nom (la clé). Cette référence (une instance du stub de l'OD) sera utilisée pour invoqué des méthodes de cet objet.

# Chargement dynamique de classes



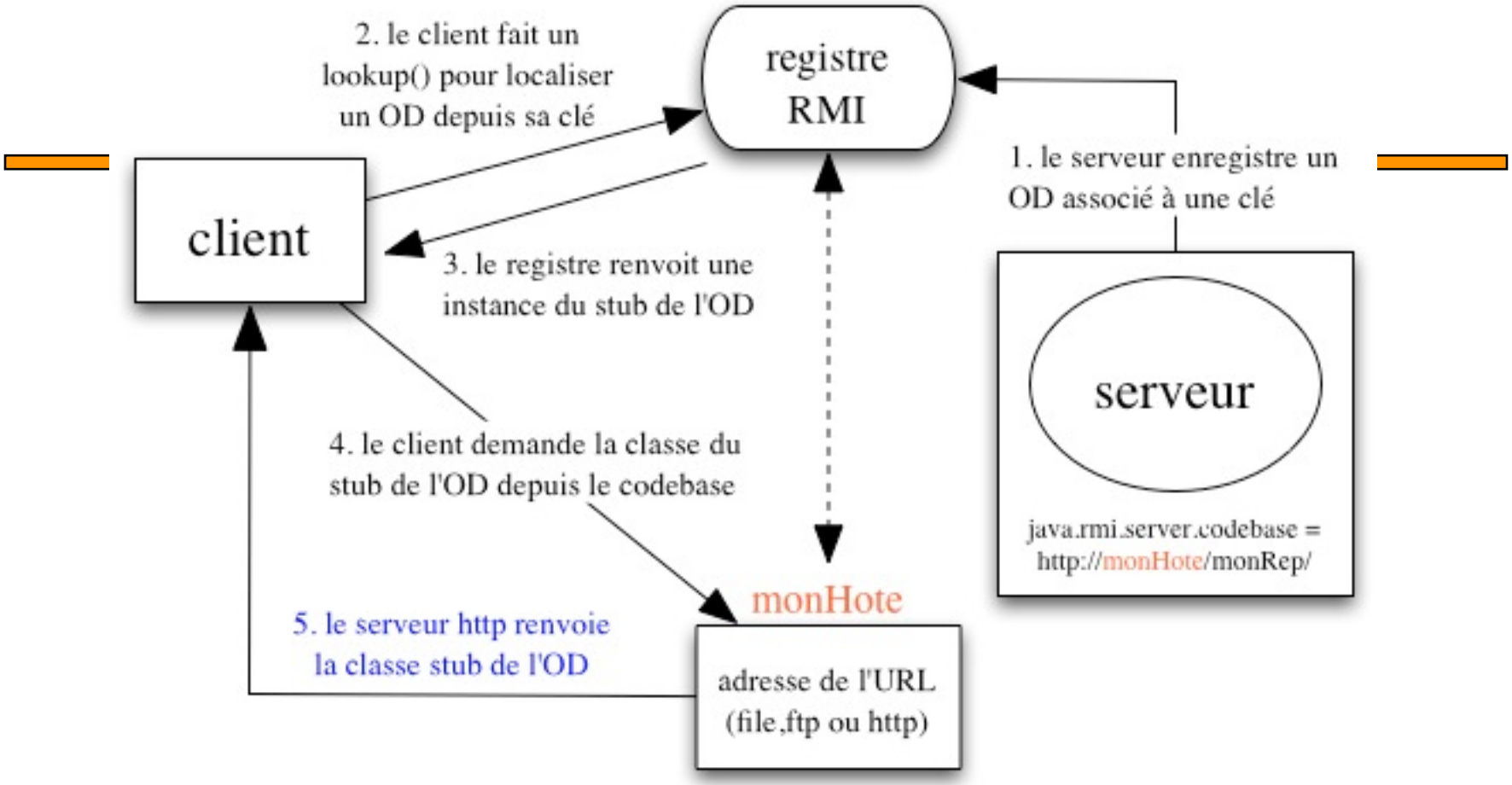

3. Le registre RMI renvoit une référence (une instance de stub) à un OD demandé. Si la définition de la classe du stub peut être trouvée localement sur le client par le CLASSPATH (toujours consulté en 1er) alors la classe est chargée localement. Cependant, si cette classe n'est pas trouvée, le client va essayer de la trouver par le codebase de l'OD.

# Chargement dynamique de classes



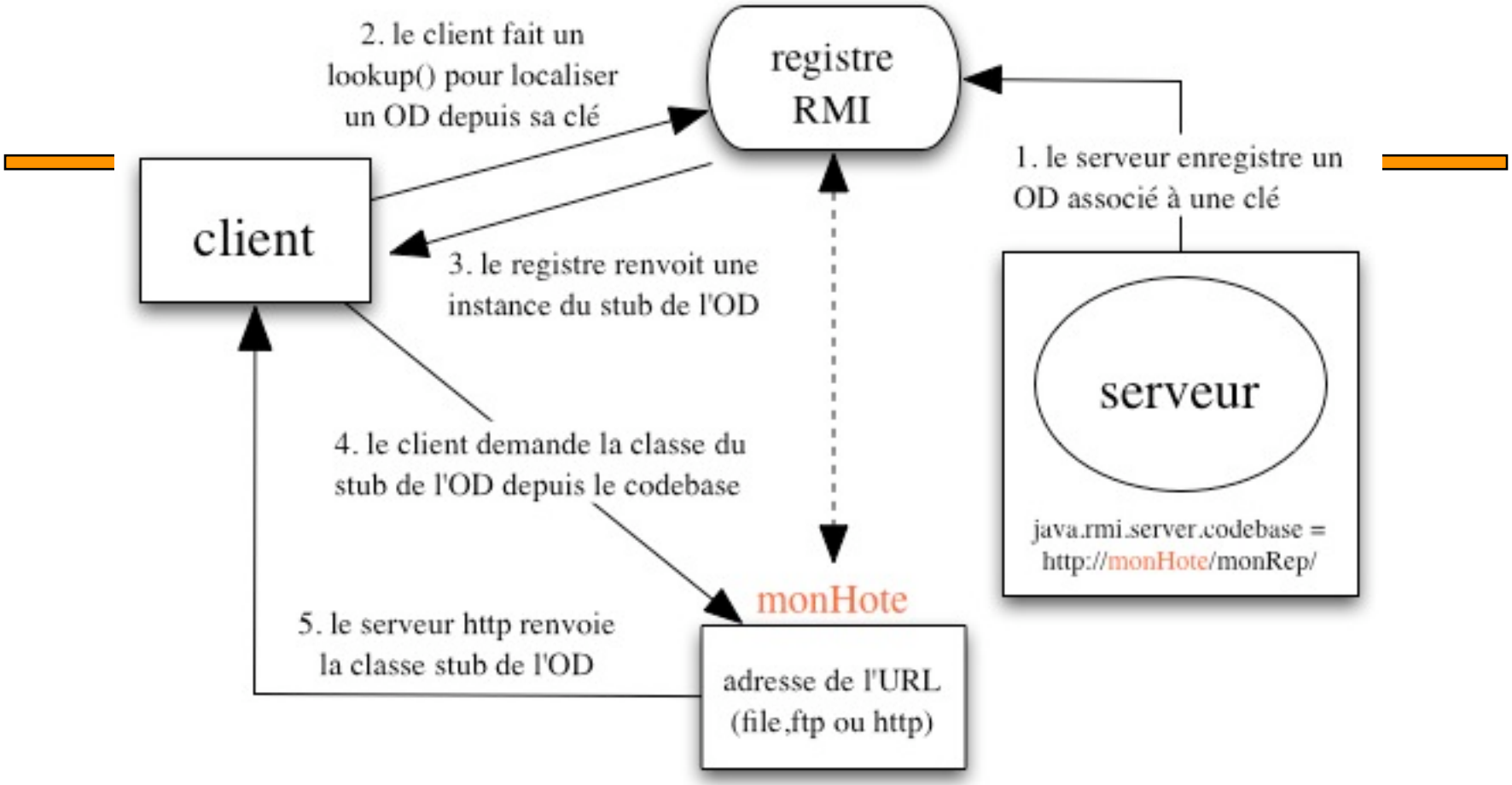

4. Le client demande la définition de la classe stub depuis le codebase.Le codebase utilisé est celui qui était spécifié pour l'OD quand la classe stub a été chargée par le registre RMI (à l'étape 1).

# Chargement dynamique de classes




5. La définition de la classe stub de l'OD est chargée sur le client depuis le codebase qui était indiqué dans le registre RMI pour cet OD

# Chargement dynamique de classes




6. Maintenant le client a toutes les informations nécessaire pour invoquer des méthodes distantes sur l'OD. L'instance de stub agit comme un proxy pour l'OD stocké côté serveur. Ainsi à la différence des applets, la méthode s'éxecute sur la machine serveur.

# Chargement dynamique de classes du client vers le serveur

- Un codebase spécifié sur le client peut être utilisé pour charger des classes distantes, locales ou des interfaces dans d'autres JVMs. Par exemple, supposons
  - Qu'une méthode d'OD attende en paramètre un objet réalisant l'interface Coupeur :

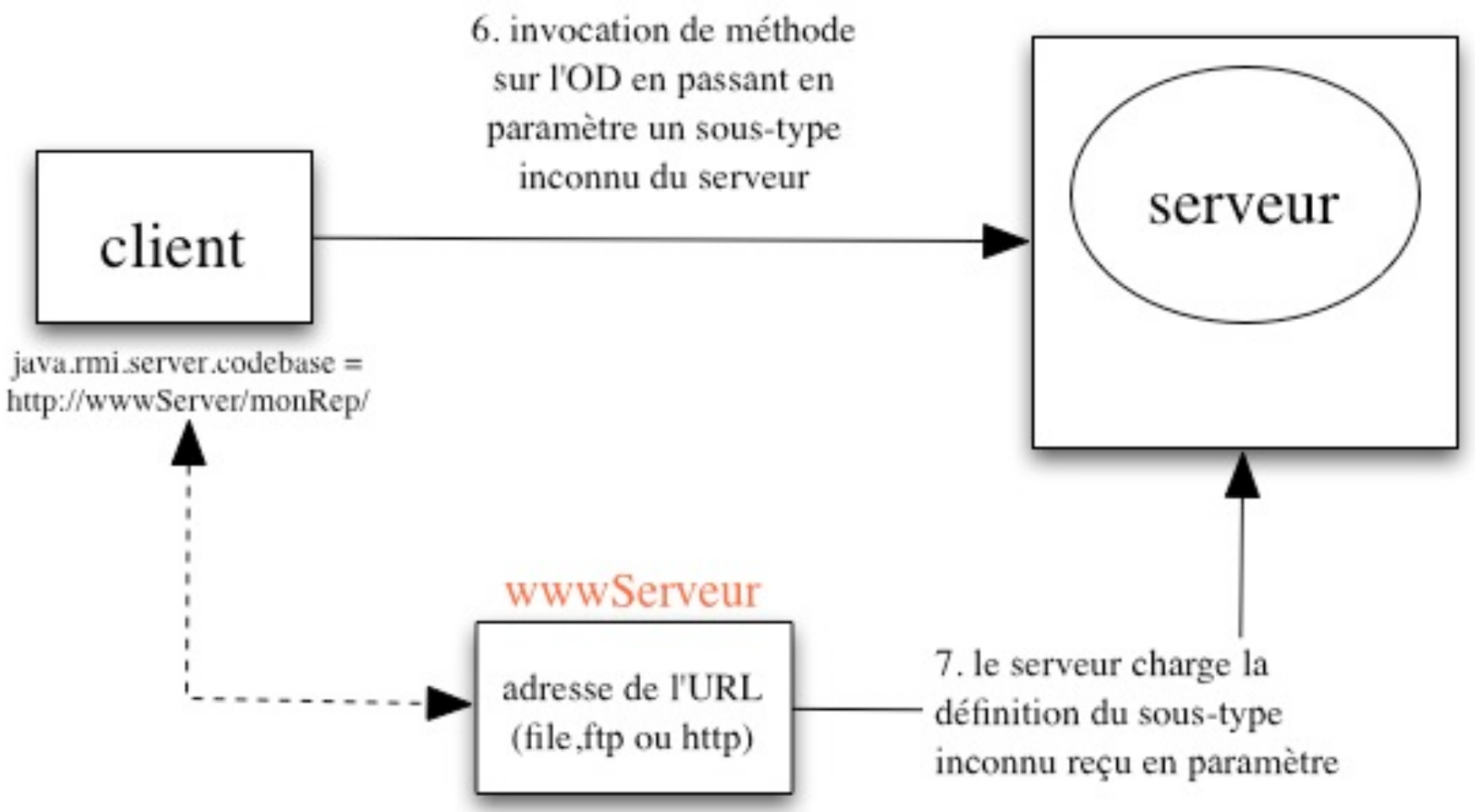
```
uneMethode(Coupeur c) { c.coupe(); … }
```

  - Que le client passe en fait en paramètre un objet de classe Scie (implémentant Coupeur),

```
OD.uneMethode(new Scie());
```

  - Que le serveur ne connaisse pas la classe Scie
- Positionner la propriété codebase dans l'application client, permet au serveur de pouvoir charger la classe Scie même s'il ne la connaît pas initialement.

# Chargement dynamique de classes du client vers le serveur



# Chargement dynamique de classe

- Dès qu'on charge des classes de l'extérieur de la JVM, des problèmes de sécurité se pose (code malveillant)
- En Java il est nécessaire de définir un gestionnaire de sécurité pour gérer les permissions de codes.
- Les package RMI proposent une classe de gestionnaire de sécurité nommé `RMISecurityManager` qui applique le règlement en vigueur sur le système.
- Ce gestionnaire de sécurité est paramétré par un fichier décrivant les permissions accordées ou refusées pour les morceaux de codes téléchargés.

# Chargement dynamique de classe

- Dans l'application chargeant le code :

```
main(…) {

 System.setSecurityManager( new RMISecurityManager() );
```

- Pour lancer l'application ensuite :

```
java -Djava.security.policy=monFichDroits
                                    MonPackage.MonAppli
```

- Syntaxe du fichier monFichDroits :

```
 grant {
          Permission java.security.AllPermission
 };
```

- Attention : la politique définie ci-dessus donne tous les droits : ne s'en servir que pour la phase de tests. Ensuite mettre en place une politique réfléchie, spécique à l'application et au rôle du code distant à charger.

# Chargement dynamique de classe

- Syntaxe du fichier monFichDroits :

```
grant {
            Permission java.security.AllPermission
};
```

- Attention : la politique définie ci-dessus donne tous les droits : ne s'en servir que pour la phase de tests. Ensuite mettre en place une politique réfléchie, spécique à l'application et au rôle du code distant à charger. Voici un exemple plus précis :

```
grant {
  permission java.net.SocketPermission "*:1024-65535",
      "connect,accept,resolve";
};
```

- Consulter la doc de Java pour connaître la syntaxe et les possibilités sur les permissions.

# Passage de paramètres

## Objets Distants :

- Lorsqu'un OD est passé d'un serveur à un client, le client reçoit en fait une instance du stub de la classe correspondant à cet OD.
- A partir de ce stub, le client peut manipuler l'OD, mais celui-ci reste sur le serveur
- Fort heureusement, on peut également passer ou renvoyer n'importe quel objet en utilisant une méthode distante, et pas seulement ceux qui implémentent Remote...

# Passage de paramètres Objets locaux

Objets Locaux (faits pour être accédés en local)

- Exemple de passage local : le programme HelloWorld renvoie un objet String
- La chaîne est créée sur le serveur et doit être amenée chez le client comme valeur de retour
- String n'implémente pas Remote
- Le client ne peut pas recevoir de stub correspondant
- Il doit se contenter d'une copie de la chaîne

# Passage de paramètres Objets locaux

- ...après l'appel, le client possède sa propre version de l'objet String, sur lequel il peut travailler.
- La chaîne reçue par le client n'a plus besoin d'aucune connexion vers un objet du serveur.

# Passage de paramètres Objets locaux

- Un objet non distant doit être transporté d'une JVM à une autre :
  - L'objet est copié
  - La copie de l'objet est envoyée au travers d'une connexion réseau
- *=> très différent d'un passage de paramètres en local (passage par référence) !!!*
  - Les références locales n'ont de sens que pour la JVM dans laquelle les objets résident (adresses)

# Passage de paramètres Objets locaux

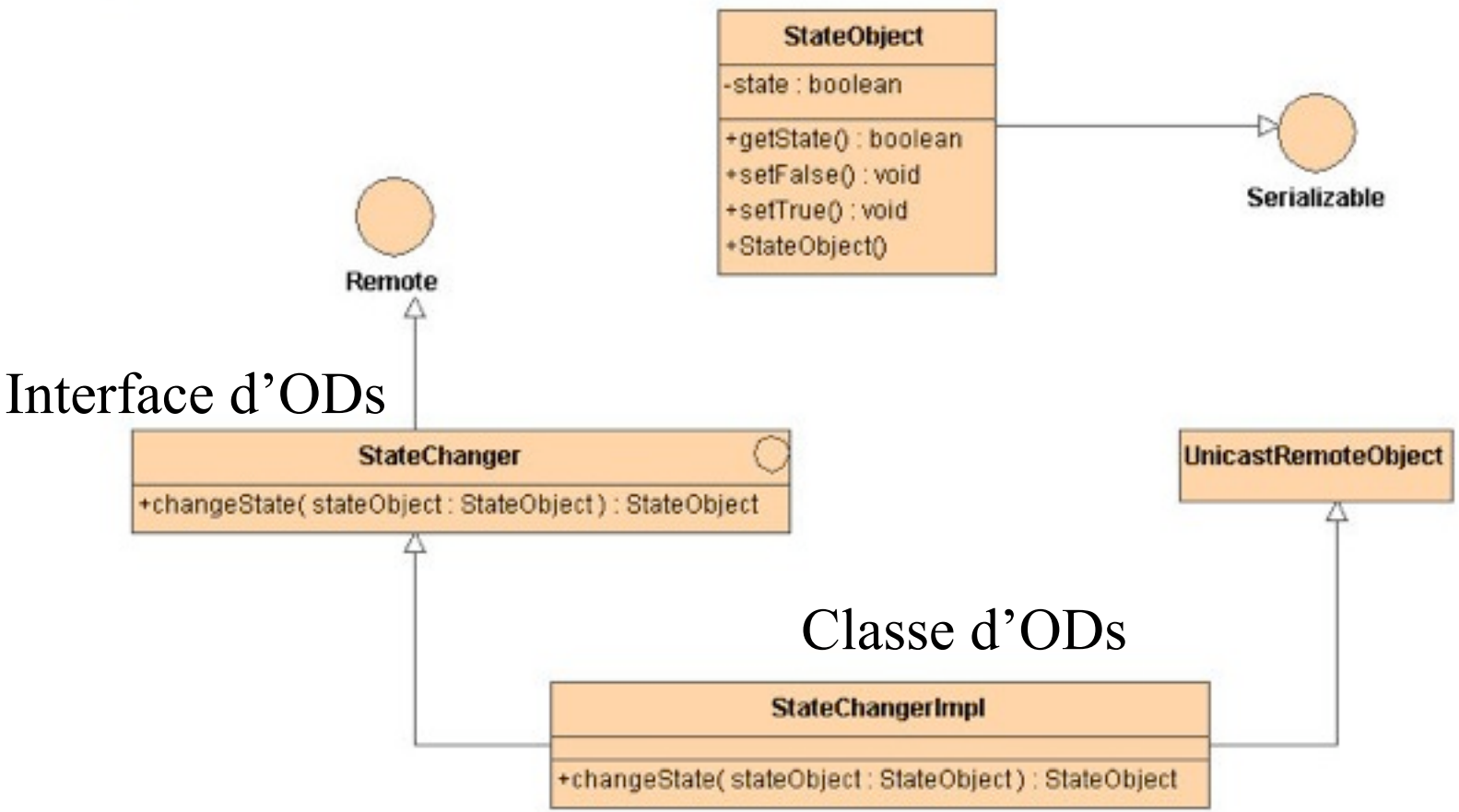
- RMI permet de transporter (copier) des objets complexes si ceux-ci ont la capacité de *se mettre en série* (pour être transmis sur le réseau)
- RMI utilise les mécanismes de *sérialisation* inclus dans Java (java.io)
- Il faut des classes d'objets implémentant l'interface Serializable

# Passage de paramètres Objets locaux - exemple

- On crée un objet sérialisable local StateObject à deux états que l'on va passer à une méthode distante
- On crée un OD avec une méthode qui change l'état d'un objet StateObject qu'on lui passe en paramètre et qui le retourne
- Le client crée un objet StateObject et affiche son état
- Il invoque la méthode du serveur en lui passant l'objet à état créé et récupère le retour dans une variable
- Il affiche l'état de l'objet référencé avant invocation, puis celui résultant de l'invocation du changement d'état

# Passage de paramètres Objets locaux - exemple

- Diagramme de classes



# Passage de paramètres Objets locaux - exemple

- La classe StateObject

```
import java.io.Serializable;

public class StateObject implements Serializable {

        private boolean state;

        public StateObject() {
                this.state = false;
        }

        public void setTrue() {
                this.state = true;
        }

        public void setFalse() {
                this.state = false;
        }

        public boolean getState() {
                return this.state;
        }
}
```

# Passage de paramètres Objets locaux - exemple

- L'interface StateChanger

```
import java.rmi.*;

public interface StateChanger extends Remote {

        public StateObject changeState(StateObject stateobject)
                throws RemoteException;

}
```

# Passage de paramètres Objets locaux - exemple

- La classe StateChangerImpl

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

public class StateChangerImpl extends UnicastRemoteObject
                              implements StateChanger {

        public StateChangerImpl()
                        throws RemoteException {
        }

        public StateObject changeState(StateObject stateObject)
                        throws RemoteException {

                if (stateObject.getState())
                        stateObject.setFalse();
                else
                        stateObject.setTrue();

                return stateObject;

        }
}
```

# Passage de paramètres Objets locaux - exemple

- Le programme serveur

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

public class StateChangerServer {

        public static void main(String[] args) {

                try {

                        System.out.println("Creating server object implementation...");
                        StateChanger changer = new StateChangerImpl();
                        System.out.println("Binding to the registry...");
                        Naming.rebind("StateChanger",changer);
                        System.out.println("Server started");

                } catch(Exception e) {
                        e.printStackTrace();
                }
        }
}
```

# Passage de paramètres Objets locaux - exemple

- Démarrage du serveur
  - On lance le rmiregistry
  - On lance le serveur

```
$ java StateChangerServer
Creating server object implementation...
Binding to the registry...
Server started
```

# Passage de paramètres Objets locaux - exemple

- Le programme client

```
import java.rmi.*;

public class StateChangerClient {

        public static void main(String[] args) {

                try {

                        // Creates a StateObject and displays its state
                        StateObject so1 = new StateObject();
                        System.out.println("so1.state = " + so1.getState());

                        // Gets a ref to the server
                        StateChanger remoteChanger =
                                (StateChanger)Naming.lookup("rmi://localhost/StateChanger");

                        // Invokes and gets the result in a new variable
                        System.out.println("invoking : so2 = remoteChanger.changeState(so1);");
                        StateObject so2 = remoteChanger.changeState(so1);

                        // Displays states for so1 and so2
                        System.out.println("so1.state = " + so1.getState());
                        System.out.println("so2.state = " + so2.getState());

                } catch(Exception e) {
                        e.printStackTrace();
                }

                System.exit(0);
        }
}
```

# Passage de paramètres Objets locaux - exemple

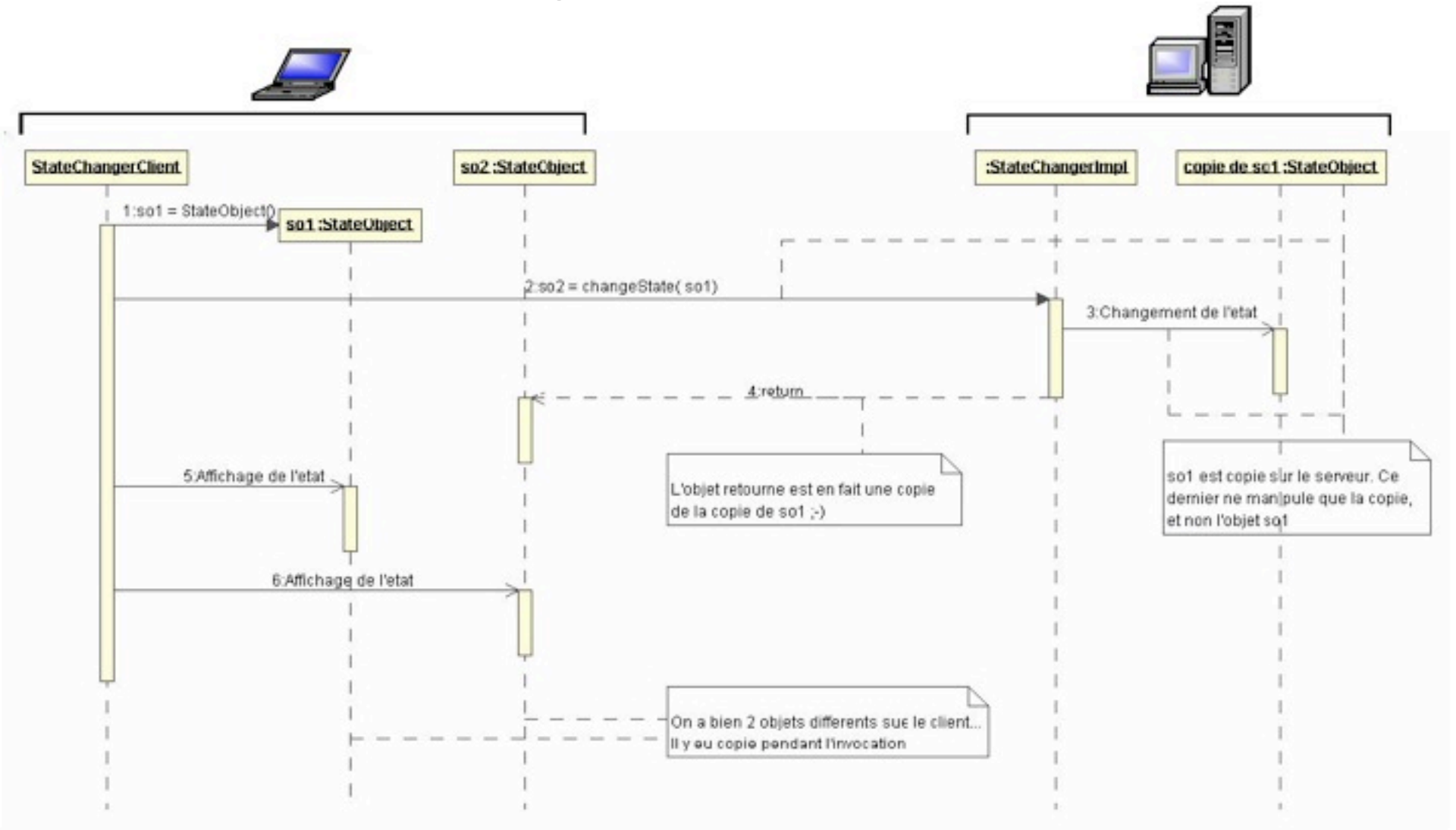
- Exécution du client

```
$ java StateChangerClient
so1.state = false
invoking : so2 = remoteChanger.changeState(so1);
so1.state = false
so2.state = true
```

- Conclusion
  - L'état de l'objet créé en local n'a pas changé, par contre, l'objet retourné a un état différent
  - Il y a bien eu copie de l'objet. Dans notre exemple, 2 copies !!!
    - Une, lors du passage de so1 en paramètre
    - L'autre, lors du retour de la valeur renvoyée par la méthode

    (mis dans so2)

# Passage de paramètres
# Objets locaux - exemple



# Passage de paramètres Objets distants

- Passage d'objets distants
  - Le passage d'un OD à une méthode ou comme valeur de retour manipule en fait un stub de l'OD
  - Exemple typique : la recherche d'objets dans la base de registres rmiregistry

```
HelloWorld h = (HelloWorld)Naming.lookup(...);
```

    - Retourne un stub pour un OD de type HelloWorld
    - L'appelant peut ensuite manipuler l'OD au travers du stub reçu
  - Pas de copie de l'objet, mais transmission du stub

# Passage de paramètres Objets distants

- Passage d'objets distants
  - Très différent du passage d'objets locaux
  - Les objets locaux sont copiés
    - Les deux protagonistes ne manipulent pas le même objet
  - Passage d'OD = passage du stub
    - Pas de copie
    - Les deux protagonistes manipulent le même objet au travers de son stub

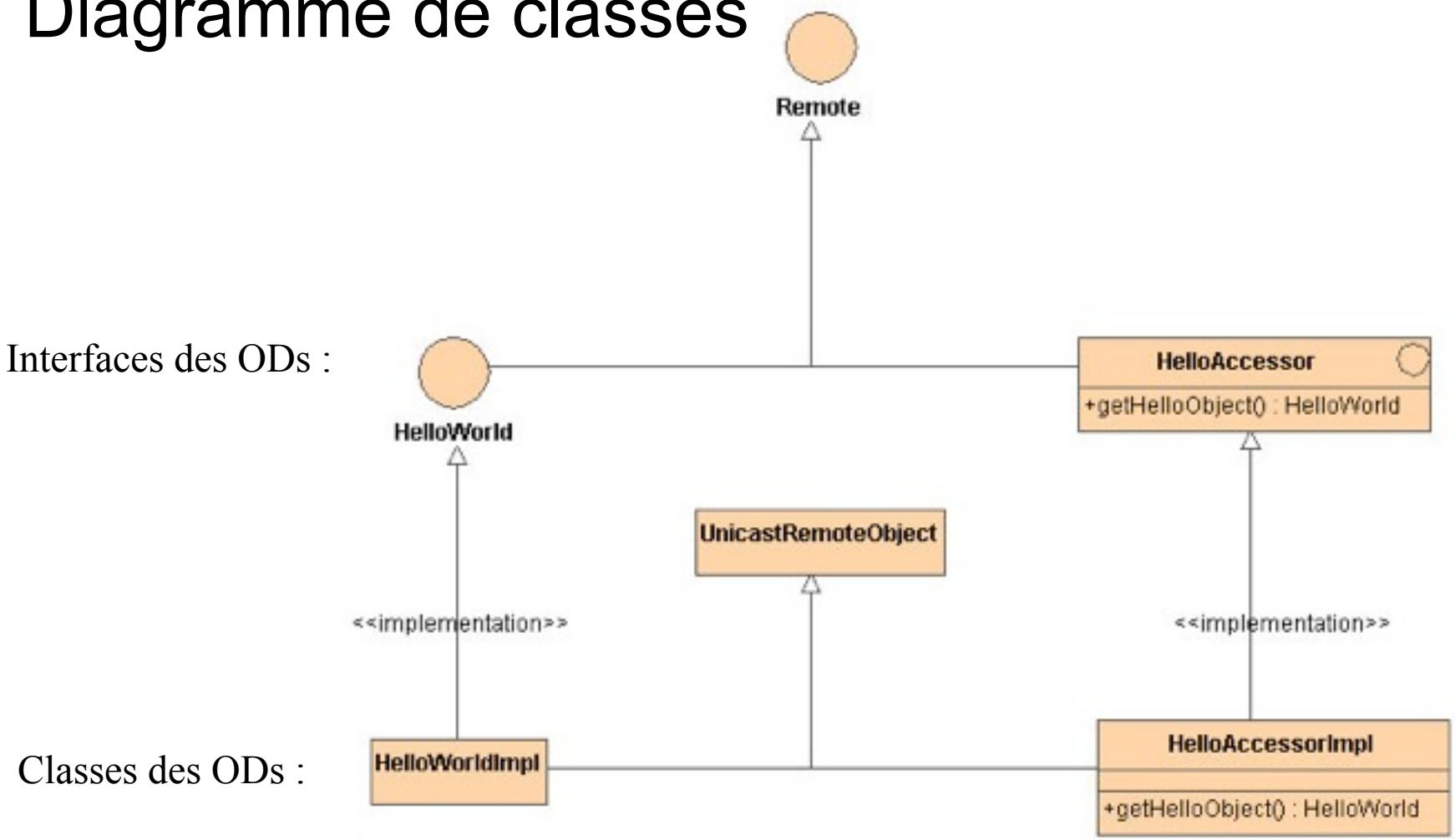
# Passage de paramètres Objets Distants

- On va créer un OD (1) de type HelloAccessor qui permet d'accéder à un autre OD (2) de type HelloWorld, situé dans la même JVM
  - Un client va
    - 1) Obtenir une référence vers l'OD (1)
    - 2) Invoquer la méthode de l'OD(1) et récupérer l'OD (2)
    - 3) Invoquer la méthode sayHello() de l'OD (2)

    => sayHello() affiche une trace dans la console... regardons si la trace s'affiche chez le client ou le serveur

# Passage d'ODs en paramètres - exemple

- Diagramme de classes



# Passage d'ODs en paramètres - exemple

- L'interface HelloAccessor

```
import java.rmi.*;

public interface HelloAccessor extends Remote {

        public HelloWorld getHelloObject()
                throws RemoteException;

}
```

# Passage d'ODs en paramètres - exemple

- La classe HelloAccessorImpl

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

public class HelloAccessorImpl
                extends UnicastRemoteObject
                implements HelloAccessor {

        private HelloWorld hello;

        // Constructor : creates HelloWorld object
        public HelloAccessorImpl() throws RemoteException {
                System.out.println("HelloAccessorImpl : creating HelloWorld object");
                this.hello = new HelloWorldImpl();
        }

        // returns the HelloWorld object
        public HelloWorld getHelloObject() throws RemoteException {
                return this.hello;
        }
}
```

# Passage d'ODs en paramètres - exemple

- Le serveur HelloAccessorServer

```
import java.rmi.*;

public class HelloAccessorServer {

        public static void main(String[] args) {

                try {

                        // Creates HelloAccessor object
                        System.out.println("Creating server implementation...");
                        HelloAccessor accessor = new HelloAccessorImpl();

                        // Binds to the registry
                        System.out.println("Binding to the registry...");
                        Naming.rebind("HelloAccessor",accessor);

                        System.out.println("Server started");

                } catch(Exception e) {
                        e.printStackTrace();
                }
        }
}
```

# Passage d'ODs en paramètres - exemple

- Démarrons le serveur
  - 1) Démarrage du rmiregistry
  - 2) Démarrage du serveur

```
$ java HelloAccessorServer
Creating server implementation...
HelloAccessorImpl : creating HelloWorld object
Binding to the registry...
Server started
```

  - => Le serveur est lancé, occupons nous maintenant du client...

# Passage d'ODs en paramètres - exemple

- Le client HelloAccessorClient

```
import java.rmi.*;

public class HelloAccessorClient {

        public static void main(String[] args) {

                try {

                        // Obtains a ref to the HelloAccessor remote object
                        HelloAccessor accessor =
                                (HelloAccessor)Naming.lookup("rmi://localhost/HelloAccessor");

                        // Obtains a ref to a HelloWorld object
                        HelloWorld hello = accessor.getHelloObject();

                        // Invokes sayHello()
                        hello.sayHello();

                } catch(Exception e) {
                        e.printStackTrace();
                }
        }
}
```

# Passage d'ODs en paramètres - exemple

- Démarrage du client

```
$ java HelloAccessorClient
```

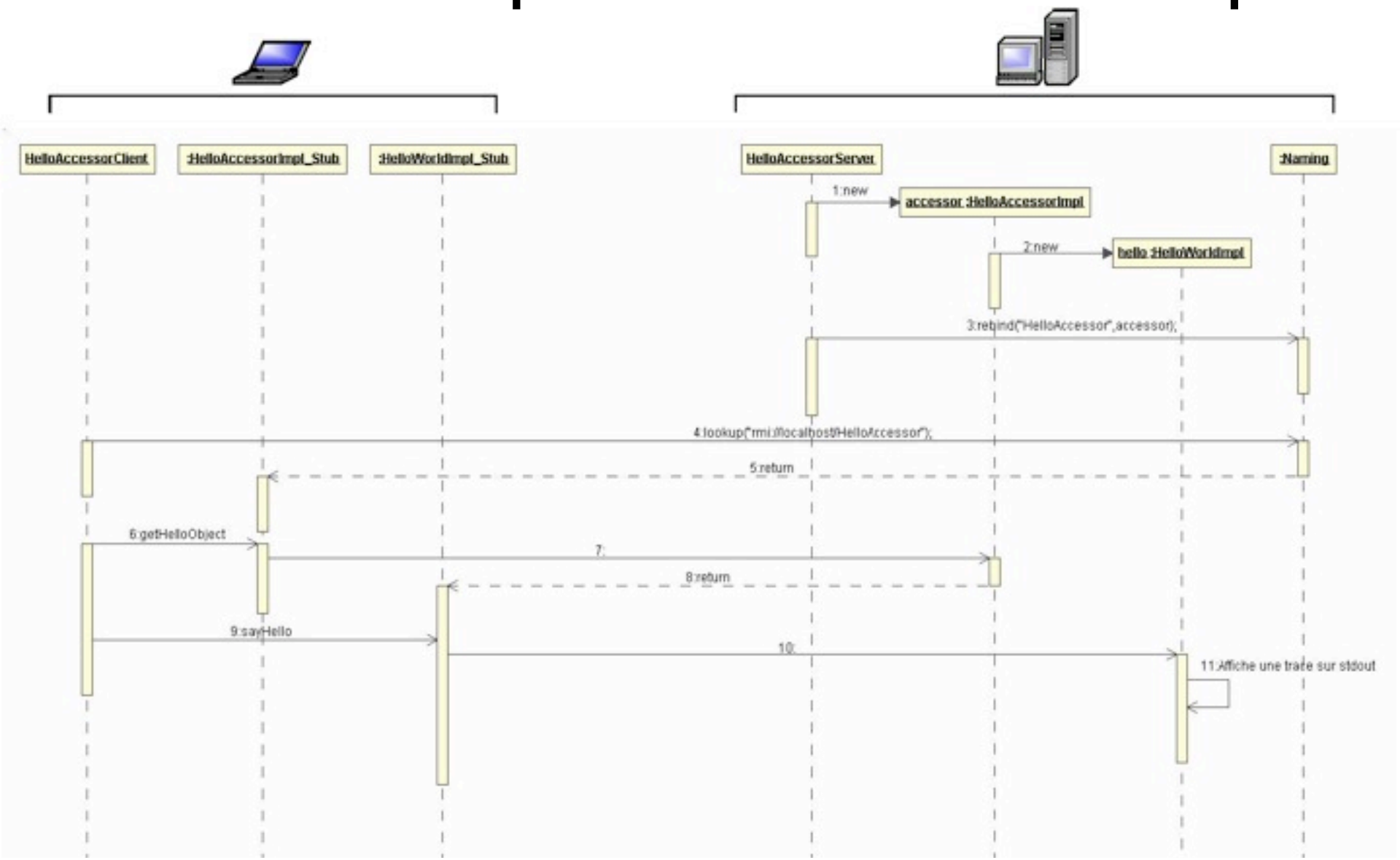
- => Pas de trace niveau client...
- Regardons la trace serveur :

```
$ java HelloAccessorServer
Creating server implementation...
HelloAccessorImpl : creating HelloWorld object
Binding to the registry...
Server started
M0thode sayHello invoqu0e...hello world !!!
```

=> La méthode a bien été exécutée sur le serveur !

Donc, le client a bien récupéré un OD (via son stub) quand il a demandé un objet HelloWorld

# Passage d'ODs en paramètres - exemple



# Passage de paramètres Paramètres inappropriés

- 1er cas : obtenir un OD graphique.
  - Pbm : l'OD est du côté serveur, donc toute méthode graphique (ex: repaint()) invoquée par le client sur un tel OD ne fait que déclencher des affichages côté serveur et non côté client !!!
- 2ème cas : passer des objets de type Graphics (java.awt) à une méthode distante
  - Graphics n'implémente pas les interfaces distantes (Remote)
  - Une **copie** de l'objet Graphics doit donc être passée
  - ...On ne peut pas faire cela !!! : en réalité, Graphics est une classe abstraite
  - Les objets Graphics sont renvoyés par un appel à Component.getGraphics()
  - Cet appel ne peut se produire que si l'on possède une sous-classe qui implémente un contexte graphique sur une plate-forme spécifiée
  - Ces objets interagissent avec le code natif et l'adressage local

# Passage de paramètres Paramètres inappropriés

- Tout d'abord, la plate-forme peut être différente
  - client sous Windows, serveur sous X11
    - Le serveur ne dispose pas des méthodes natives nécessaires pour afficher des graphiques sous Windows
    - Pas un soucis si on travaille avec Swing et non AWT…
- Mais, même si la plate-forme est la même (ex: client et serveur sous X11), les objets graphiques référencent des blocs mémoires situés à des adresses spécifiques à l'exécution en cours sur la machine serveur

  => copier ces pointeurs vers le client ne servirait à rien : de ce côté là, ces adresses n'ont pas de sens (la mémoire du client est dans un état différent de celle du serveur).

**Conclusion** *: **copier** un objet de type Graphics n'a pas de sens. Ces objets ne sont donc pas sérialisables et ne peuvent être envoyés par RMI*

# Passage de paramètres Conclusion

- Il faut être vigilant quand au passage des paramètres dans les applications RMI...
  - Certains objets sont copiés (les objets locaux), d'autres non (les objets distants)
  - Les ODs, non copiés, résident toujours sur le serveur
  - Les ODs sont manipulés côté client par l'intermédiaire de *stubs* de façon transparente pour le programmeur
  - Les objets locaux passés en paramètre doivent être sérialisables afin d'être copiés, et ils doivent être indépendants de la plate-forme
    - Les objets qui ne sont pas sérialisables lèveront des exceptions
    - Attention aux objets sérialisables qui utilisent des ressources locales !!!

# Conclusion

- RMI permet aux objets Java et à leurs méthodes de devenir distants relativement facilement
- Le JDK inclut les interfaces et classes spéciales pour RMI par défaut
- On peut passer presque tous les types de paramètres
- Les objets de serveur peuvent être accédés en environnement concurrent

Nous avons une bonne réponse a notre question initiale => Voilà une bonne façon de distribuer des objets !!!

# De la doc pour aller plus loin...

- **Core Java Volume 2**
  - Par Cay S. Horstmann & Gary Cornell, Editions CampusPress
  - Une référence pour les développeurs Java et contient une bonne section sur RMI (base de ce cours)
- Java Distributed Computing
  - Par Jim Farley, Editions O'Reilly
  - Tout sur les applications reparties avec Java (plus technique…)
- Site de Sun :
  - http://java.sun.com/docs/books/tutorial/rmi/