Chapitre 3

RMI : les bases

L'objectif de ce chapitre est d'apprendre la réalisation d'applications distribuées avec RMI de JAVA

1. INTRODUCTION 2

2. PRESENTATION DES PACKAGES RMI 2

2.1. LE PACKAGE JAVA.RMI.* 2
2.1.1. La classe Naming 2
2.1.2. La classe RMISecurityManager 3
2.1.3. L'interface Remote 3
2.1.4. Les exceptions 3
2.2. LE PACKAGE JAVA.RMI.SERVER.* 3
2.2.1. La classe UnicastRemoteObject (URO) 3
2.3. LE PACKAGE JAVA.RMI.REGISTRY.* 4
2.3.1. La classe LocaleRegistry 4
2.3.2. L'interface Registry 5

3. LE COMPILATEUR IDL 5

4. L'ADAPTATEUR D'OBJET (OU ANNUAIRE) 5

4.1. PAR UNE CLASSE « MAIN » 5
4.2. PAR UNE COMMANDE PREDEFINIE 6
4.3. PAR LE CODE 6
4.4. LES PRINCIPES 6

5. LE SOCKET FACTORY RMI 7

6. CREATION D'UN OD AVEC RMI 8

6.1. CREATION PAR HERITAGE 8
6.2. CREATION PAR CONSTRUCTION 8

7. MISE EN ŒUVRE 8

8. EXEMPLE SIMPLE : HELLO 10

9. EXEMPLE DE LA CREATION D'UN OD : UNE DEVISE 12

10. EXEMPLE D'UN FACTORY DE DEVISE 17

1. Introduction

RMI = Remote Method Invocation

RMI est une solution Java permettant de réaliser l'invocation à distance de méthode d'objet.

Cette technologie s'insère dans les principes des architectures à Base de Composants.

Elle est issue de la technologie RPC (Remote Process Call) déjà utilisée dans les années 70 (norme d'échange XDR) permettant d'appeler un traitement entre deux process informatiques.

Dans le cadre du cours NSY102, RMI est utilisé pour la mise en œuvre des Designs Patterns dans les corrections des exemples et exercices.
La seule manière de comprendre et prouver le fonctionnement d'un DP est de le coder. Nous avons fait le choix du langage Java et de RMI.

Il ne vous sera jamais demandé de réaliser de codage dans le cadre du cours NSY 102. Mais comme tous les exemples de la mise en œuvre des DP de ce cours sont réalisés en JAVA, il est impératif que vous soyez à jour dans la compréhension de cette technologie RMI.

2. Présentation des packages RMI

Les API Java prédéfinies permettant la mise en œuvre d'application RMI sont :

- java.rmi
- java.rmi.server
- java.rmi.registry

*2.1. Le package java.rmi.**

Ce package définit les classes, interfaces et exception utilisées par l'appelé et l'appelant des services d'un OD.

2.1.1. La classe Naming

Cette classe gère le service de nommage créé par la classe LocalRegistry (voir plus bas).
Observez que ces méthodes sont des méthodes statiques.

- static void **bind**(String name, Remote obj)
  Binds the specified name to a remote object.
- static String[] **list**(String name)
  Returns an array of the names bound in the registry.
- static Remote **lookup**(String name)
  Returns a reference, a stub, for the remote object associated with the specified name.

- static void **rebind**(String name, Remote obj)
  Rebinds the specified name to a new remote object.
- static void **unbind**(String name)
  Destroys the binding for the specified name that is associated with a remote object

2.1.2. La classe RMISecurityManager

Cette classe permet d'initialiser la politique de sécurité à prendre en compte.

En salle de TP ou en restant en local de votre poste, vous n'avez pas besoin de gérer la politique de sécurité.

2.1.3. L'interface Remote

Toutes les interfaces de méthodes distantes d'un nouvel OD doivent héritées de cette interface prédéfinie. C'est ce qui permet à Java de savoir que la méthode est une méthode distante.

2.1.4. Les exceptions

RemoteException
Toute méthode distante est susceptible de déclencher cette exception.
Elle doit donc être propagée lors de la création des méthodes distantes.

AlreadyBoundException et NotBoundException
Cas d'erreur sur les opérations RMI (bind, lookup, …)

*2.2. Le package java.rmi.server.**

Ce package est utilisé par le programme qui crée un **Objet Distant** (ou serveur).

2.2.1. La classe UnicastRemoteObject (URO)

Cette classe est la classe principale, le coeur de la creation des OD en RMI.

Le principe : tout objet dont la classe d'appartenance **hérite** de cette classe est un OD.

Les constructeurs :

- protected **UnicastRemoteObject**()
  Creates and exports a new UnicastRemoteObject object using an anonymous port.
- protected **UnicastRemoteObject**(int **port**)
  Creates and exports a new UnicastRemoteObject object using the particular supplied port.
- protected **UnicastRemoteObject**(int port, RMIClientSocketFactory csf, RMIServerSocketFactory ssf)
  Creates and exports a new UnicastRemoteObject object using the particular supplied port and socket factories.

Les méthodes :

- static RemoteStub **exportObject**(Remote obj)
  Exports the remote object to make it available to receive incoming calls using an anonymous port.

- static Remote **exportObject**(Remote obj, int port)
  Exports the remote object to make it available to receive incoming calls, using the particular supplied port.
- static Remote **exportObject**(Remote obj, int port, RMIClientSocketFactory csf, RMIServerSocketFactory ssf)
  Exports the remote object to make it available to receive incoming calls, using a transport specified by the given socket factory.
- static boolean **unexportObject**(Remote obj, boolean force)
  Removes the remote object, obj, from the RMI runtime.

Cette classe hérite de **RemoteServer** qui hérite de **RemoteObject**. Les méthodes très utililes sont :

public static String getClientHost()

public static Remote toStub(Remote obj)
Cette méthode est très importante : elle permet d'obtenir le stub d'un Objet Distribué. Un stub est un Proxy de Communication.

*2.3. Le package java.rmi.registry.**

Ce package est consacré à la gestion de l'annuaire RMI : création et utilisation de l'annuaire.

2.3.1. La classe LocaleRegistry

Cette classe permet de créer, par le logiciel, le service de nommage RMI.

- static Registry **createRegistry**(int port)
  Creates and exports a Registry on the local host that accepts requests on the specified port.
-
- static Registry **createRegistry**(int port, RMIClientSocketFactory csf, RMIServerSocketFactory ssf)
  Creates and exports a Registry on the local host that uses custom socket factories for communication with that registry.
- static Registry **getRegistry**()
  Returns a reference to the the remote object Registry for the local host on the default registry port of 1099.
- static Registry **getRegistry**(int port)
  Returns a reference to the the remote object Registry for the local host on the specified port.
- static Registry **getRegistry**(String host)
  Returns a reference to the remote object Registry on the specified host on the default registry port of 1099.
- static Registry **getRegistry**(String host, int port)
  Returns a reference to the remote object Registry on the specified host
- port.static Registry **getRegistry**(String host, int port, RMIClientSocketFactory csf)
  Returns a locally created remote reference to the remote object

Registry on the specified host and port.Il existe d'autres méthodes que nous utiliserons pas dans un premier temps.

Toutes ces méthodes retournent une interface d'utilisation de l'annuaire.

2.3.2. L'interface Registry

Les méthodes permettant d'utiliser un annuaire RMI..

void **bind**(String name, Remote obj)
Binds a remote reference to the specified name in this registry.
String[] **list**()
Returns an array of the names bound in this registry.
Remote **lookup**(String name)
Returns the remote reference bound to the specified name in this registry.
void **rebind**(String name, Remote obj)
Replaces the binding for the specified name in this registry with the supplied remote reference.
void **unbind**(String name)
Removes the binding for the specified

3. Le compilateur IDL

En RMI, le compilateur IDL est une fourniture du JDK java que l'on trouve dans le répertoire bin (exemple sur le JEE 1.5 : C:\Sun\AppServer\jdk\bin)

Il est donc accessible via le PATH

*La syntaxe : **rmic** <package>.<nom classe OD>*

La commande rmic permet de générer les squelettes (skelton) et les amorces (stub) RMI.
<nom classe OD> est le nom de la classe de l'OD (qui hérite de UnicastRemoteObject et surtout hérite d'une interface dans laquelle sont décrites les méthodes distantes de l'OD

Il est à noter que depuis la version 1.5, il est inutile d'utiliser la commande rmic car java prend en charge nativement et dynamiquement la création des objets de classes Stub et Skeleton.
Le stub est créé grâce au DP **DynamicProxy**.

En RMI-IIOP, il est indispensable d'utiliser la commande rmic (rmic –iiop …).

Egalement, si vous devez utiliser la méthode exportObject pour créer par agrégation un OD, alors il faut utiliser la commande rmic afin de générer les stubs et les skeltons.

4. L'adaptateur d'objet (ou annuaire)

Il existe 3 façons d'exécuter un adaptateur RMI.

4.1. Par une classe « main »

En utilisant la classe prédéfinie sun.rmi.registry.RegistryImpl ;

java -classpath "packages" sun.rmi.registry.RegistryImpl 9100

avantages :

- un meilleur pilotage du classpath, se lance à l'extérieur

inconvénients :

- doit toujours être lancé avant tous les autres programmes serveur, peut être obsolète (confusion avec java.rmi.registry.RegistryImpl), le port est un paramètre de commande OS

4.2. Par une commande prédéfinie

rmiRegistry 9100

avantages :

- façon récente d'exécuter un adaptateur à l'extérieur

inconvénients :

- pas de pilotage du classpath, variable d'environnement, le port est un paramètre de commande OS

4.3. Par le code

```
try {LocateRegistry.createRegistry(9100);}
     catch(Exception l_ex){};
```

avantages :

- intégrer au code, le port peut être précisé par configuration (fichier, properties, …),
- S'il existe déjà alors pas de création d'un annuaire supplémentaire.

inconvénients :

- intégrer au code, fonction du classpath de la JVM, nécessite un programme serveur maître devant être exécuté avant tous les autres. Mais cela est souvent le cas.

4.4. Les principes

On passe en paramètre du lancement le numéro de port sur lequel les clients devront se connecter.

Ce numéro doit être libre (voir votre administrateur système). Pour vos applications sur vos PC vous pouvez prendre par exemple un numéro élevé 9100.

On peut exécuter plusieurs adaptateurs sur une même machine du moment qu'ils s'exécutent sur des ports différents

L'adresse de connexion à un adaptateur rmi est : **rmi://host:port**
où

- host est l'adresse IP de la machine sur lequel s'exécute l'adaptateur
- port est le port utiliser par l'adaptateur
- rmi est le nom du protocole de connexion utilisé (on peut ne pas le mettre: par défaut rmi)

Exemple :
Sur le client : je suis sur la machine pc-10 :

```
o = Naming.lookup("rmi://pc-03:9100/TOTO");
```

Ou

```
o = Naming.lookup("rmi://localhost:9100/TOTO")
```

Et sur le serveur :

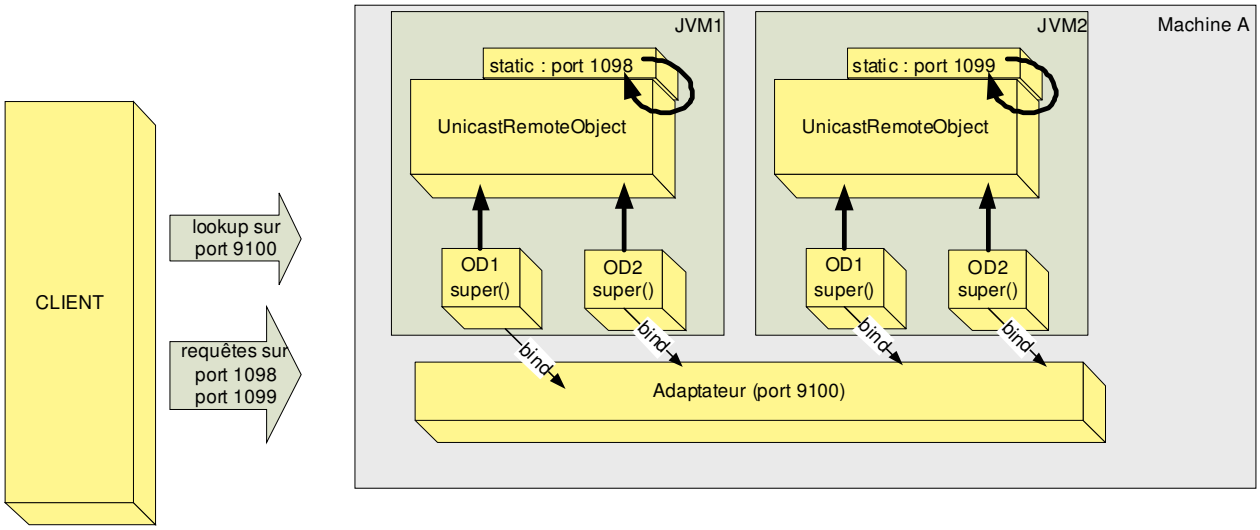
```
Naming.rebind("rmi://localhost:9100/TOTO",od);
```

5. Le Socket Factory RMI

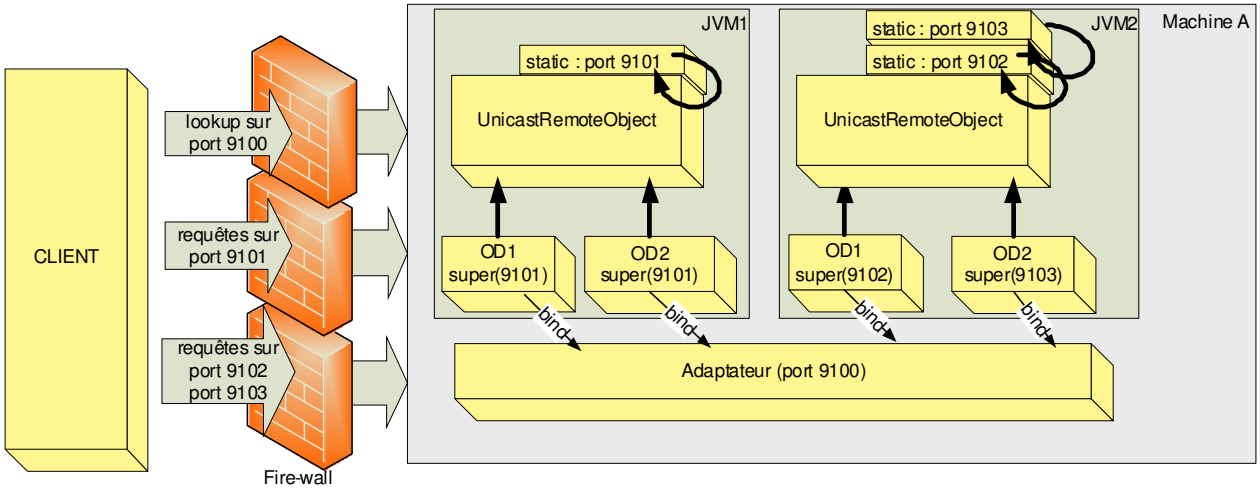
Le Socket Factory de RMI permettant de traiter les requêtes des clients est de niveau "static"

Le port d'écoute est :

- soit déterminée par défaut (le premier libre),
  - Le même port pour une même JVM
  - Un par JVM
  - utilisation de **super()**
  - les ports sont alloués par le système automatiquement
  - pas de politique ciblé de sécurité
  - Un seul serveur de socket par JVM



- soit imposé
  - UnicastRemoteObject super(port)
  - le port doit être libre
  - on peut associer plusieurs ports
  - permet de sécuriser l'utilisation des objets distribués via les firewalls
  - utilisation de **super(port)**
  - Autant de serveur de socket que de port différent
  - Attention au conflit de port pour une même machine



Commentaires :

- dans la JVM1 tous les objets distribués s'exécutent sur le même port (9101)
- dans la JVM2 chaque objet distribué s'exécute sur un port différent

6. Création d'un OD avec RMI

Il existe deux façons pour créer un OD : par héritage ou par construction.

On privilégie la première car plus facile à mettre en place et même si une classe que l'on veut transformer en OD qui hériterait déjà d'une classe, **il vaut mieux par conception créer une nouvelle classe qui encapsule la classe cible**. Il n'est pas utile que toutes les méthodes soient exportables. De plus, souvent, on a besoin de créer de nouveaux attributs spécifiques à la mise en place de RMI.

6.1. Création par héritage

Héritage de la classe **UnicastRemoteObject**.

Mettre dans son constructeur : super() ou super(port)

6.2. Création par construction

Utilisation de la méthode **exportObjet** de **UnicastRemoteObject**.

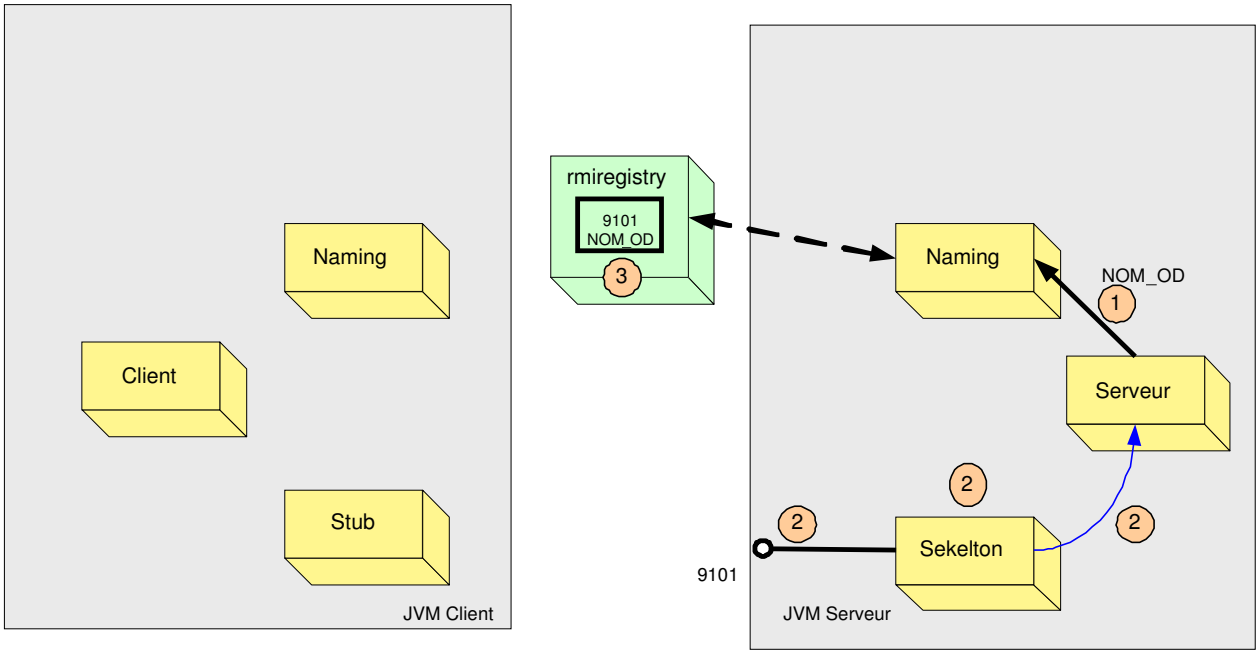
Possède les mêmes signatures que les constructeurs avec en plus l'objet qui implémente les méthodes distantes:

- public static RemoteStub exportObject(Remote obj, …)

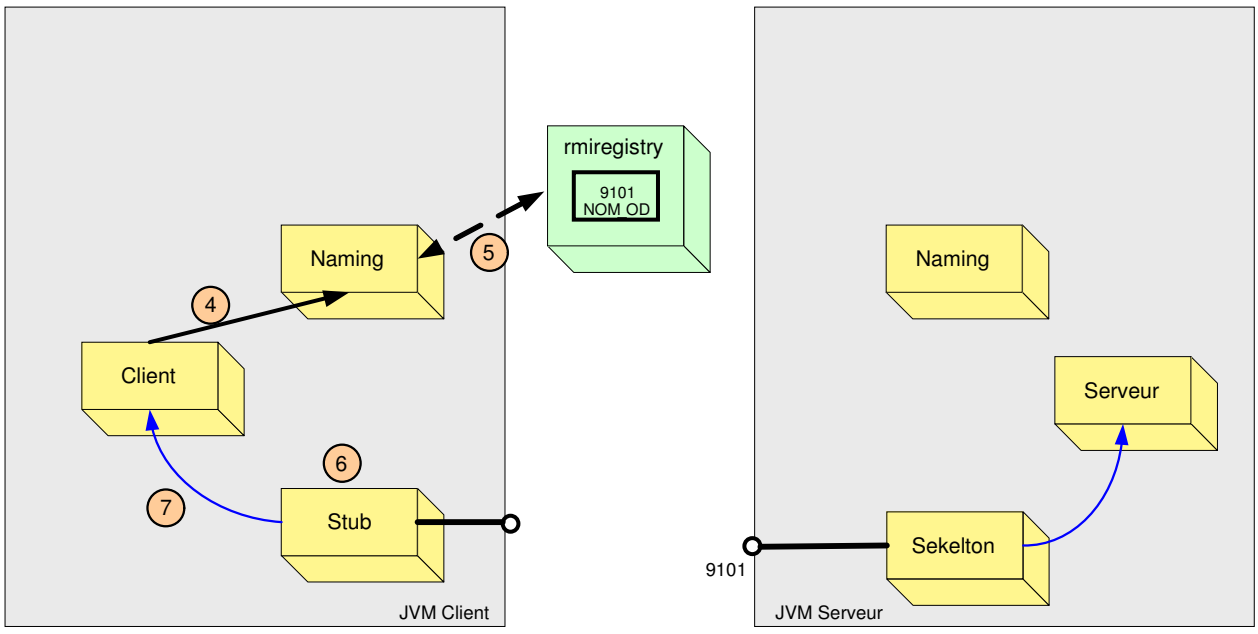
Attention : dans ce cas, il faut générer le stub avec la commande rmic.

On verra un exemple plus loin.

7. Mise en œuvre

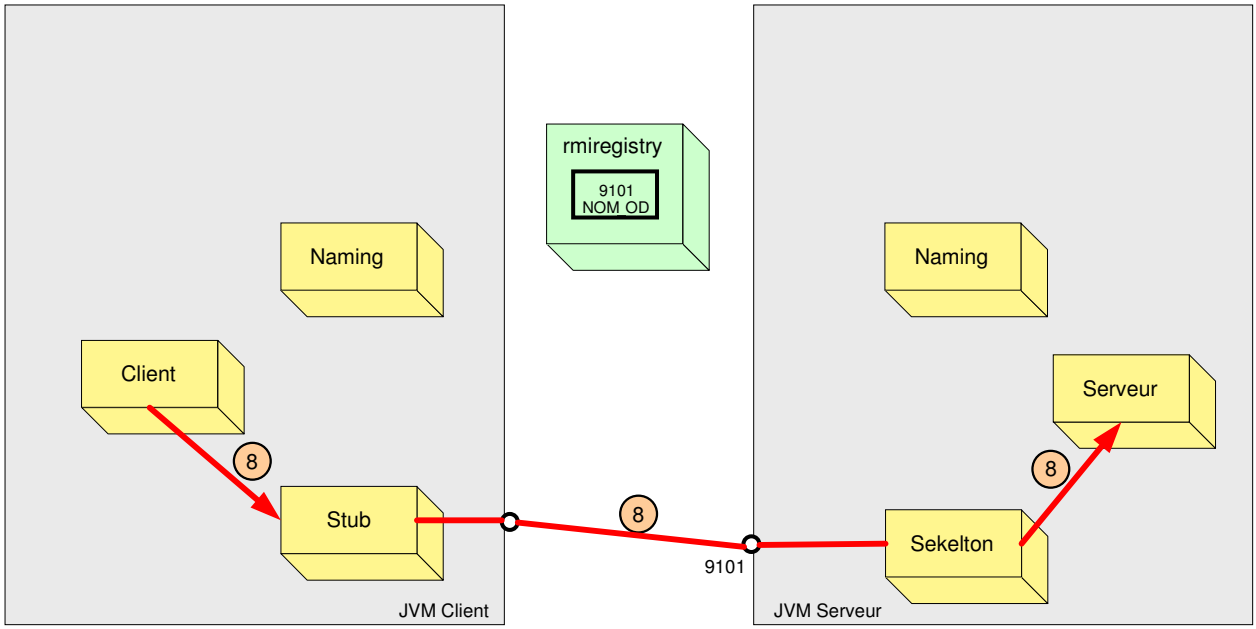


- 1 - L'objet serveur s'enregistre auprès du Naming de sa machine (méthode rebind)
- 2 - L 'objet skeleton est créé par le Naming (ou a été généré par la commande rmic). Le serveur de socket du serveur se met en attente de socket de communication sur le port et le skelton maintient une référence vers l'objet serveur
- 3 - Le Naming enregistre le stub dans le service de nommage



- 4 - L'objet client fait appel au Naming pour obtenir le stub (méthode lookup)
- 5 - Le Naming récupère le stub,

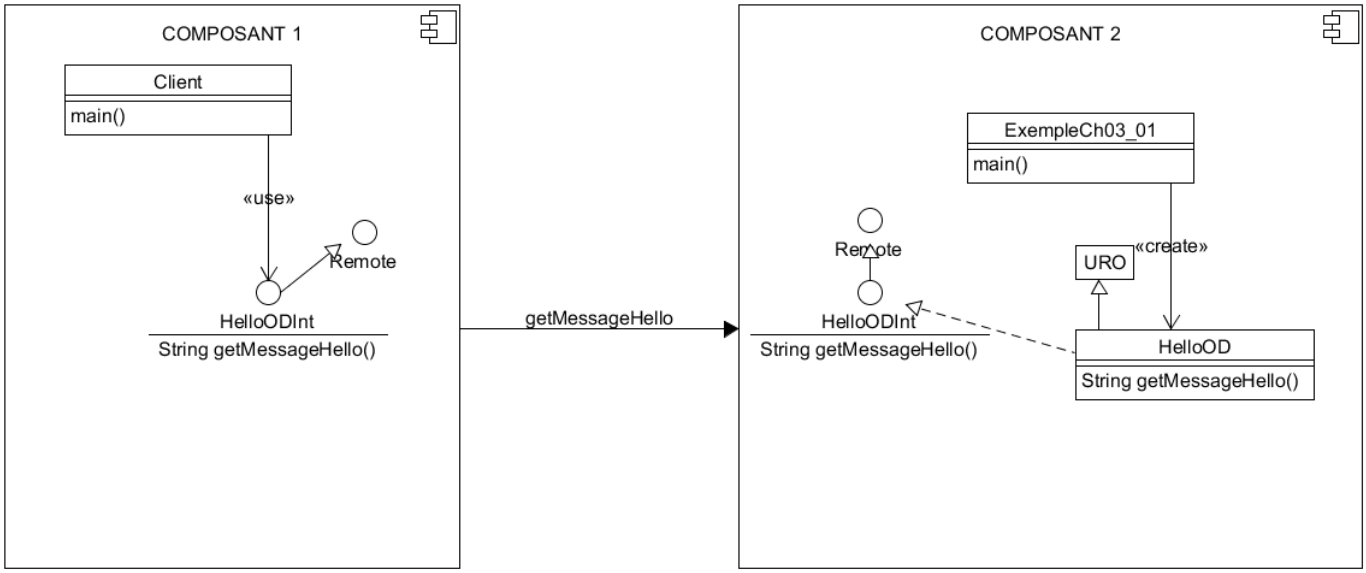
- 6 – et le stub fait une demande d'ouverture de socket
- 7 – maintient une référence au client
- 8 - Le client effectue l'appel de méthode distante par appel à l'objet Stub. Les requêtes transitent via le socket.



8. Exemple simple : HELLO

EXEMPLE Java

Cet exemple est disponible sur le site : ExempleCh03_01_Hello



```
  // Exemple simple en RMI : le message HELLO

  // ExempleCh03_01.java
  //
  import java.lang.*;
  import java.io.*;

  // Les packages indispensables pour utiliser RMI
  import java.rmi.*;
  import java.rmi.registry.*;
  import java.rmi.server.*;

  // Le programme serveur qui crée l'OD
  public class ExempleCh03_01
  {
      public static void main(String args[]) throws Exception
      {
          // Création du service de nommage utilisé par RMI
          try{
              LocateRegistry.createRegistry(9100);
          } catch(Exception ex){};

          // Création de l'objet distribué
          System.out.println("Création de l'objet distribué");
          HelloOD od = new HelloOD(9101,
                                   "Bonjour, je m'appelle Pierre DUPONT");

          // Enregistrement de l'OD dans l'annuaire sous le nom "HELLO"
          //  Ce nom doit être unique dans l'annuaire
          System.out.println("Enregistrement de l'objet distribué");
          Naming.rebind("rmi://localhost:9100/HELLO",od);

          /* Un server de socket est instancié et est exécuté en fond par
             un thread. Ce qui explique que si on ne fait pas les lignes
qui suivent
             le programme ne s'arrête pas.

             Pour terminer proprement, il faut tuer proprement tous les OD,
ce qui
             provoquera la fin du serveur de socket
          */

          System.out.println("Bus en ecoute....");
          // L'opérateur décide l'arrêt de l'OD
          DataInputStream in = new DataInputStream(System.in);
          System.out.print("Taper rc, pour arreter le serveur...");
          System.out.flush();
          String valeur= in.readLine();

          // Désenregistrement de l'OD de l'annuaire
          Naming.unbind("rmi://localhost:9100/HELLO");

          // Destruction de l'OD
          UnicastRemoteObject.unexportObject(od,true);   //   true   :   pour
forcer

      }
  }
```

```
  // HelloOD.java
```

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

  public class HelloOD extends UnicastRemoteObject implements HelloODInt
  {
      private String messageHello;

      public HelloOD(int portServerSocket,
                     String messageHello) throws RemoteException
      {
          super(portServerSocket);   // Le port utilisé par le server de
socket
          this.messageHello = messageHello;
      }

      public String getMessageHello()
      {
          return(messageHello);
      }
  }
```

```
  // HelloODInt.java
  //
  import java.rmi.*;

  public interface HelloODInt extends Remote
  {
      public String getMessageHello() throws RemoteException;
  }
```

```
  // Client.java

  import java.rmi.*;

  public class Client
  {
    public static void main(String args[]) throws Exception
    {
        String hostServeur="localhost";
        if (args.length==1)
            hostServeur=args[0];

      HelloODInt bonjour =
(HelloODInt)(Naming.lookup("rmi://"+hostServeur+":9100/HELLO"));

       while(true)
          {
              String message = bonjour.getMessageHello();
              System.out.println(message);
              try{Thread.sleep(500);}catch(Exception ex){};
          }
  }
```

9. Exemple de la création d'un OD : une devise

Cet exemple consiste à créer sur un serveur 2 OD. Chacun contient le service de conversion d'une devise vers une autre.

Ces 2 OD sont utilisés depuis un programme client.

Il est possible de changer le taux d'une devise depuis le serveur.

Cet exemple est disponible sur le site : ExempleCh03_02_Devise

Diagramme de classe du serveur :

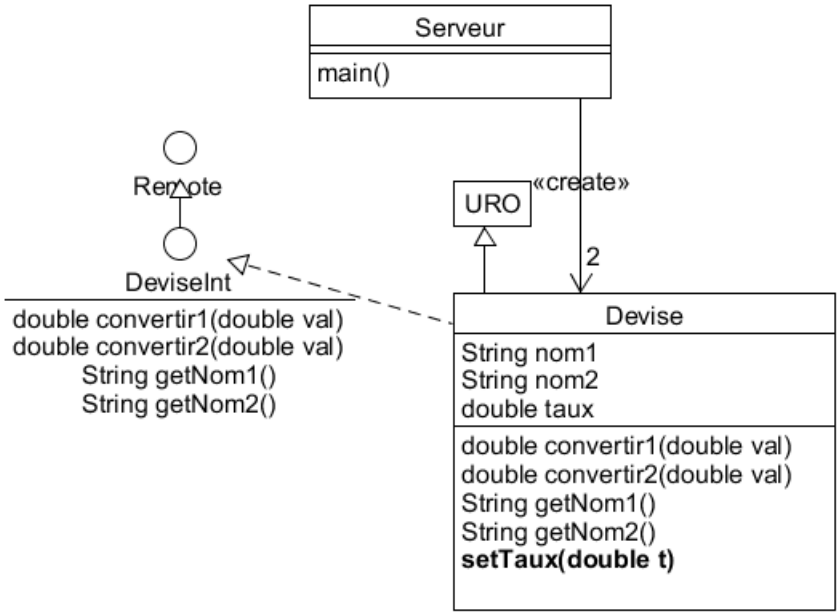
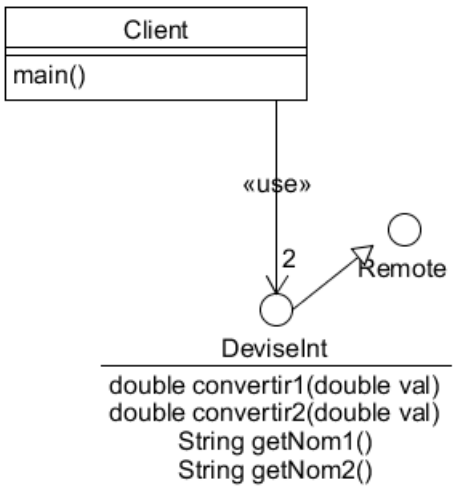


Diagramme de classe du client :




```
// Serveur.java
// Le serveur cree deux objets distribues de devise :
//     FRANCS/EURO et FRANCS/DOLLARS
// Il permet aussi de changer dynamiquement le taux de la
//    devise FRANCS/DOLLARS
//
import java.io.*;
```

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

  /* Les packages de RMI */
  import java.rmi.*;
  import java.rmi.server.*;
  import java.rmi.registry.*;

  public class Serveur
  {
      public      static      void      main(String      args[])      throws
RemoteException,ServerNotActiveException,MalformedURLException, IOException
      {
          // Creation de l'annuaire s'il n'existe pas déjà
          try{
              LocateRegistry.createRegistry(9100);
          } catch(Exception ex){};

          // Creation de deux objets distribués
          // 1 pour chaque conversion de monnaie
          //
          Devise devise1 = new Devise("FRANCS","EURO",1.0/6.56);
          Naming.rebind("rmi://localhost:9100/FRANCS_EURO",devise1);

          Devise devise2 = new Devise("FRANCS","DOLLARS",0.98);
          Naming.rebind("rmi://localhost:9100/FRANCS_DOLLARS",devise2);

          System.out.println("Services enregistres pour le serveur");

          /* On permet d'intervenir sur le serveur */
          while (true)
              {
                  DataInputStream in = new DataInputStream(System.in);
                  System.out.print("Nouveau taux pour francs-dollars: ");
                  System.out.flush();
                  double t = Double.parseDouble(in.readLine());
                  devise2.setTaux(t);
              }
      }
  }
```

```
  // Client.java
  // Le client demande la saisie des montants monetaires et les convertit
  //   en utilisant les méthodes distantes des objets distribues
  //
  import java.awt.*;
  import java.io.*;
  import java.net.*;

  /* Les packages de RMI */
  import java.rmi.*;
  import java.rmi.server.*;

  public class Client
  {
    public      static      void      main(String      args[])      throws
RemoteException,IOException,NotBoundException
    {
        // Adresse Ip ou nom de domaine du serveur
        String hostServeur="localhost";
```

```
        if (args.length==1)
            hostServeur=args[0];

        // Connexions aux deux devises créées sur le serveur
        DeviseInt francs_euro = (DeviseInt)Naming.lookup(
                        "rmi://"+hostServeur+":9100/FRANCS_EURO");
        DeviseInt francs_dollars = (DeviseInt)Naming.lookup(
                        "rmi://"+hostServeur+":9100/FRANCS_DOLLARS");

        // Programme de test d'utilisation des deux devises
        while (true)
          {
              double val,resultat;
              DataInputStream in = new DataInputStream(System.in);

              System.out.println("Conversion francs en euro");
              System.out.print("Francs: ");
              System.out.flush();
              val = Double.parseDouble(in.readLine());
              resultat = francs_euro.convertir1(val);
              System.out.println("Euro: "+resultat);

              System.out.println("Conversion euro en frans");
              System.out.print("Euro: ");
              System.out.flush();
              val = Double.parseDouble(in.readLine());
              resultat = francs_euro.convertir2(val);
              System.out.println("Franc: "+resultat);


              System.out.println("Conversion francs en dollars");
              System.out.print("Francs: ");
              System.out.flush();
              val = Double.parseDouble(in.readLine());
              resultat = francs_dollars.convertir1(val);
              System.out.println("Dollars: "+resultat);

              System.out.println("Conversion dollars en francs");
              System.out.print("Dollars: ");
              System.out.flush();
              val = Double.parseDouble(in.readLine());
              resultat = francs_dollars.convertir2(val);
              System.out.println("Franc: "+resultat);
          }
    }
}
```

```
// Devise.java
// Classe de definition d'un objet distribue de Devise dont le role
//   est d'offrir les services de conversion monetaire  par deux
//   méthodes distantes (dans les 2 sens)
// Une devise est caracterisee par deux noms de monnaie et un
//   taux de conversion.
//

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

```
/* Les packages de RMI */
import java.rmi.*;
import java.rmi.server.*;

public class Devise extends UnicastRemoteObject implements DeviseInt
{
  private String nom1;
  private String nom2;
  private double taux;

  public Devise(String n1,String n2,double t) throws RemoteException
  {
    super(9101);
    nom1=n1;
    nom2=n2;
    taux=t;
  }

  public void setTaux(double t)
  {
    taux=t;
  }

  public double convertir1(double val) throws RemoteException
  {
      System.out.println("Appel de convertir 1");
    return(val*taux);
  }

  public double convertir2(double val) throws RemoteException
  {
    return(val/taux);
  }

  public String getNom1()  throws RemoteException
  {
    return(nom1);
  }

  public String getNom2()  throws RemoteException
  {
    return(nom2);
  }

}
```

```
// DeviseInt.java
// Interface de l'objet distribue Devise
// L'interface permet de convertir dans les deux sens les monnaies et
//  permet de connaitre les noms des monnaies de la devise
//

import java.rmi.*;

public interface DeviseInt extends Remote
{
  public double convertir1(double val) throws RemoteException;
  public double convertir2(double val) throws RemoteException;
  public String getNom1() throws RemoteException;
  public String getNom2() throws RemoteException;
```

```
}
```

10. Exemple d'un factory de devise

Cet exemple est disponible sur le site : ExempleCh03_03_FactoryDevise

Diagramme de communication de cet exemple :

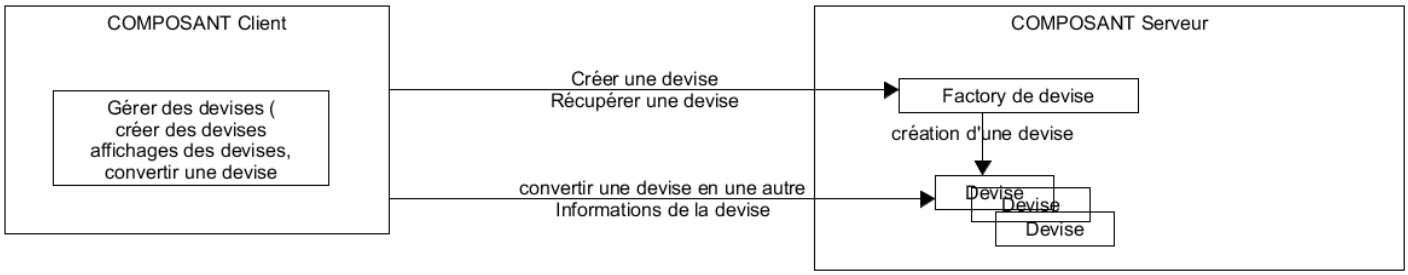


Diagramme de classe du composant Serveur :

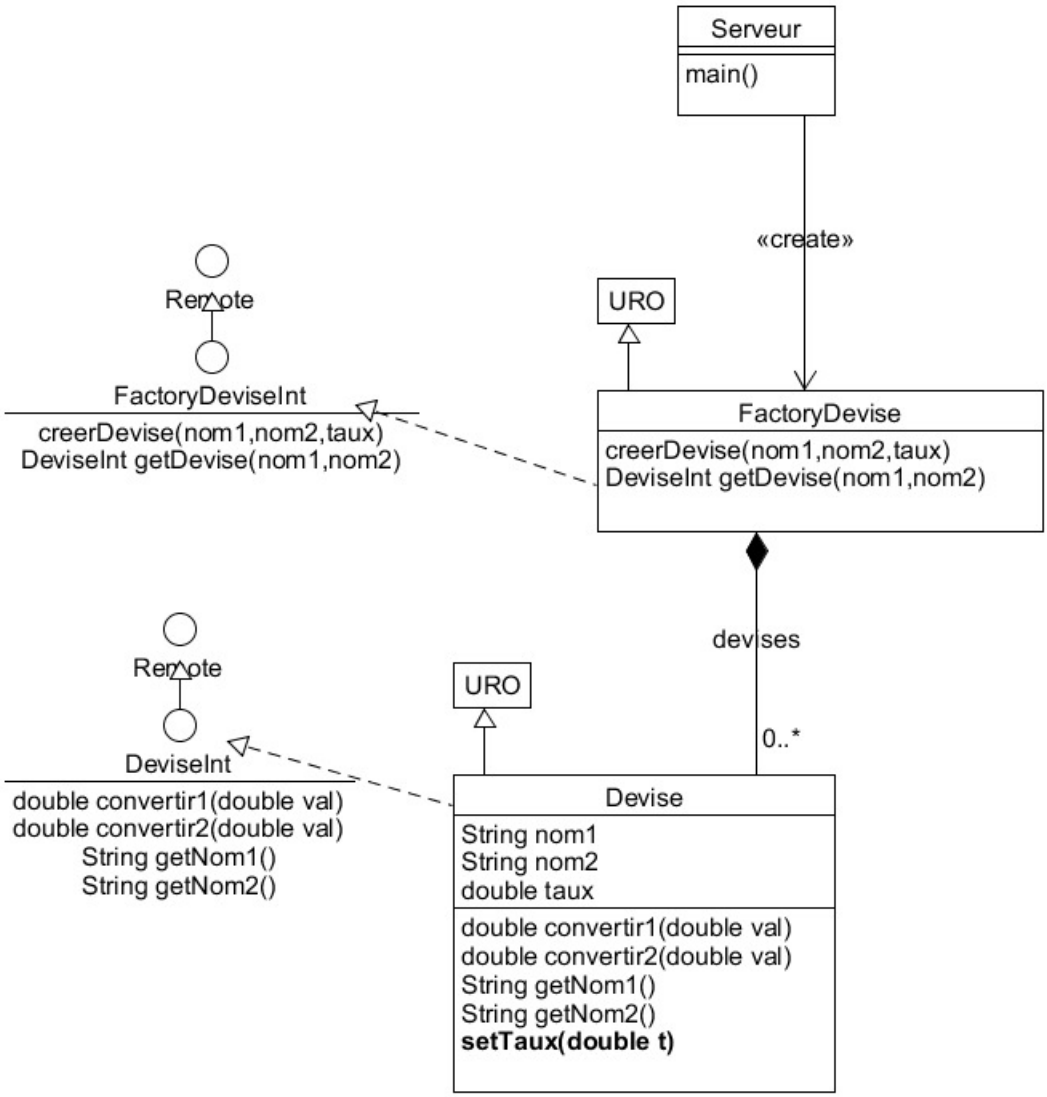
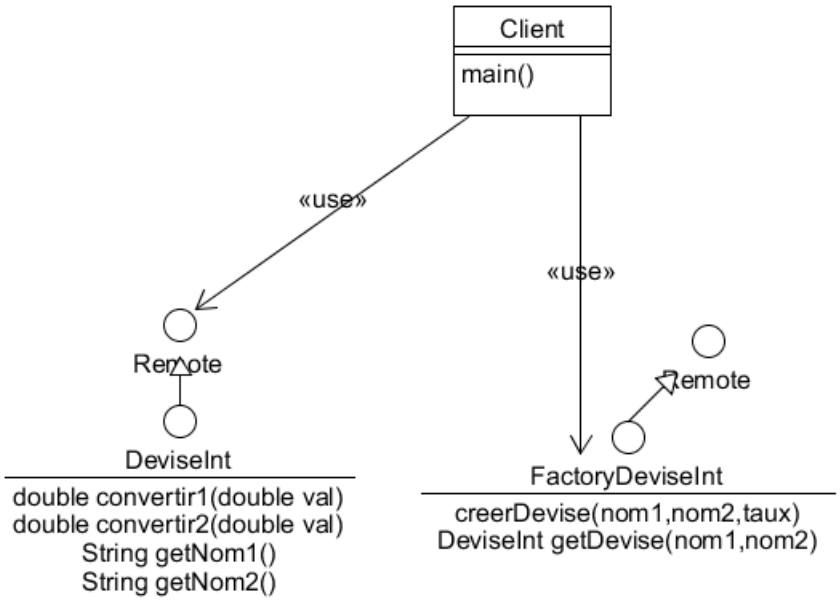


Diagramme de classe du composant Client :



Codage de cet exemple :

```
  // FactoryDevise.java
  // Le factory de devise est un OD qui permet de créer a distance
  //   une nouvelle devise qui est aussi un OD
  // Il permet aussi d'obtenir le stub de n'importe quelle devise.
  //

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

  /* Les packages de RMI */
  import java.rmi.*;
  import java.rmi.server.*;

  public class FactoryDevise extends UnicastRemoteObject implements
FactoryDeviseInt
  {
      // Stockage des OD qui sont des devises indexées par les
      //    noms de monnaie
      //
      private Hashtable<String,Devise> devises;

      // Constructeur
      public FactoryDevise() throws RemoteException
      {
          super(9101);
          devises = new Hashtable<String,Devise>();
      }

      // Creation distante d'une devise
      public void creerDevise(String nom1, String nom2,double taux) throws
RemoteException
      {
          System.out.println("Creation d'une devise: "+nom1+" "+nom2);
```

```
            Devise devise = new Devise(nom1,nom2,taux);
            devises.put(nom1+"_"+nom2,devise);
        }

        // Retourne le stub d'une devise pour le client
        public DeviseInt getDevise(String nom1, String nom2) throws
RemoteException
        {
            try{
                Devise devise = devises.get(nom1+"_"+nom2);
                Remote stub = UnicastRemoteObject.toStub((Remote)devise);
                return((DeviseInt)stub);
            }catch(NoSuchObjectException ex){return null;}
        }
    }
```

Remote stub = UnicastRemoteObject.toStub((**Remote**)devise);

Un stub est un Proxy.

```
    // FactoryDeviseInt.java
    // L'interface du factory qui definit ses services

    import java.rmi.*;

    public interface FactoryDeviseInt extends Remote
    {
        public void creerDevise(String nom1, String nom2,double taux) throws
RemoteException;
        public DeviseInt getDevise(String nom1, String nom2) throws
RemoteException;
    }
```

```
    // Serveur.java
    // Le serveur cree l'annuaire, cree le factory et l'enregistre dans
l'annuaire
    //

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

    /* Les packages de RMI */
    import java.rmi.*;
    import java.rmi.server.*;
    import java.rmi.registry.*;

    public class Serveur
    {
        public static void main(String args[]) throws
RemoteException,ServerNotActiveException,MalformedURLException, IOException
        {
            // Creation de l'annuaire s'il n'existe pas déjà
            try{
```

```
            LocateRegistry.createRegistry(9100);
        } catch(Exception ex){};

        // Creation du factory de devise
        //
        Naming.rebind("rmi://localhost:9100/DEVISES",
                      new FactoryDevise());
    }
}
```

```
// Client.java
// Le client utilise le factory distant pour :
//   - creer deux nouvelles devises par defaut
//   - permet de créer une nouvelle devise et la tester
//

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

/* Les packages de RMI */
import java.rmi.*;
import java.rmi.server.*;

public class Client
{
    public      static      void      main(String      args[])      throws
RemoteException,IOException,NotBoundException
    {
        // Adresse Ip ou nom de domaine du serveur
        String hostServeur="localhost";
        if (args.length==1)
            hostServeur=args[0];

        // Connexion au factory de devise
        FactoryDeviseInt                      fact                      =
(FactoryDeviseInt)Naming.lookup("rmi://"+hostServeur+":9100/DEVISES");

        // Création à distance de deux devises par défaut
        fact.creerDevise("FRANCS","EURO",1.0/6.56);
        fact.creerDevise("FRANCS","DOLLARS",0.98);

        // Programme de test de creation de devise et
        //  d'utilisation des devises
        //
        while (true)
            {
                double val,resultat;
                DataInputStream in = new DataInputStream(System.in);

                System.out.println("Creer un devise (o,n) ?");
                String rep = in.readLine();
                if (rep.equals("o"))
                    {
                        System.out.print("Nom1 : ");
                        String nomd1 = in.readLine();
                        System.out.print("Nom2 : ");
                        String nomd2 = in.readLine();
                        System.out.print("Taux : ");
                        double taux = Double.parseDouble(in.readLine());
```

```
                        fact.creerDevise(nomd1,nomd2,taux);
                    }

                System.out.println("Conversion de ");
                String nom1 = in.readLine();
                System.out.println("vers ");
                String nom2 = in.readLine();

                DeviseInt devise = fact.getDevise(nom1,nom2);

                System.out.print(nom1+" : ");
                System.out.flush();
                val = Double.parseDouble(in.readLine());
                resultat = devise.convertir1(val);
                System.out.println(nom2+" : "+resultat);
            }
        }
    }
```

Illustration d'un Factory distant :