

# Java Avancé - Cours 2

## Plan

<b>1</b>	<b>Communication entre objets</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Relations entre le panier et le rayon . . . . .	1
1.3	Remote Interface . . . . .	2
1.4	Le rayon fruits et légumes . . . . .	2
1.5	Le panier . . . . .	3
<b>2</b>	<b>Ant</b>	<b>4</b>
2.1	Description de Ant . . . . .	4
2.2	Hiérarchie . . . . .	4
2.3	Compilation avec Ant - le tag javac . . . . .	4
2.4	Exécuter rmic - le tag rmic . . . . .	5
2.5	Exécuter le serveur - le tag java . . . . .	5
2.6	Effacer les fichiers générés . . . . .	6
2.7	Invoquer Ant . . . . .	6
<b>3</b>	<b>Exécuter les clients et serveurs RMI</b>	<b>7</b>
3.1	rmiregistry . . . . .	7
3.2	Lancer le serveur . . . . .	7
3.3	Lancer le client . . . . .	7

## 4 Exercices

7

Le but de ce cours est de détailler un mécanisme de communication directement entre des objets java distants. Cette méthode nommée RMI (Remote Method Invocation) permet qu'un objet java donné puisse appeler une méthode d'un objet java situé sur une autre machine.

## 1 Communication entre objets

### 1.1 Motivation

Aujourd'hui un serveur Web possède une architecture qui peut être complexe. Quand google me donne une réponse à ma recherche, le serveur Web qui me répond ne possède pas en local sur son disque les 4 milliards de pages. Il communique donc avec d'autres machines.

Java étant facilement utilisé dans les serveurs Web, il est dès lors important de pouvoir communiquer directement entre les objets.

Le but de cette section est de réaliser un exemple où deux objets communiquent entre eux:

- un objet `Panier`, qui contient un entier représentant le nombre de pommes dans le panier,
- un objet `Rayon` qui contient un certain nombre de pommes, et qui possède une méthode permettant de retirer ou d'ajouter une pomme.

### 1.2 Relations entre le panier et le rayon

Il nous faut tout d'abord définir les relations entre le panier dans lequel nous mettons les pommes et le rayon dans lequel nous les prenons.

- le panier envoie au rayon un objet permettant de savoir combien de pommes il faut prendre,
- le rayon reçoit cet objet et renvoie pour information le nombre de pommes restantes dans l'étalage.

### 1.3 Remote Interface

La communication entre le client et le serveur (e.g., le panier et le rayon) est codé par deux classes constituant la remote interface. Nous mettons les classes de la remote interface dans un paquetage nommé `boutique`.

Voici tout d'abord la définition du rayon d'un magasin, il possède deux méthodes permettant d'ajouter ou de retirer des pommes.

```
package boutique;                // les classes vont être définies dans boutique.*

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Rayon extends Remote {
    Object ajoutePomme(ChangeRayon t) throws RemoteException;
    Object retirePomme(ChangeRayon t) throws RemoteException;
}
```

Tout d'abord la déclaration `package boutique` indique que les déclarations à venir vont être faites dans le package `boutique`. Ainsi pour utiliser ce qui va être défini il faudra utiliser un commande `import boutique.*` ou bien référencer explicitement les noms.

On constate ensuite l'usage du mot clé `interface` au lieu de `class` dans la déclaration de `Rayon` cela signifie que `Rayon` n'est pas une classe dont on va pouvoir directement créer des instances en faisant `new Rayon`. Il faut voir l'interface comme un contrat, ainsi en écrivant l'interface `Rayon` il suffit de donner le prototype des méthode et non leur corps.

Voici ensuite les données que l'on fournit à l'interface `boutique.Rayon`, qui donne le nombre d'articles que l'on souhaite prendre :

```
package boutique;

import java.io.Serializable;

public interface ChangeRayon extends Serializable {
    Object nombreDArticles();
}
```

`ChangeRayon` hérite de `Serializable`. Cela signifie que les instances de `ChangeRayon` peuvent être transformées en une suite de caractères permettant de sauvegarder l'objet. C'est cette sauvegarde qui va être l'information qui va circuler entre les machines.

### 1.4 Le rayon fruits et légumes

Nous allons maintenant construire un rayon précis, le rayon fruits et légumes qui va devoir respecter l'interface précédente. Cette classe sera dans un paquetage nommé `mesrayons`:

```
package mesrayons;

import boutique.ChangeRayon;
import boutique.Rayon;

public class FruitsEtLegumes
    extends java.rmi.server.UnicastRemoteObject
    implements boutique.Rayon
{
    int nombreDePommes=0;

    public FruitsEtLegumes() throws java.rmi.RemoteException {
        super();
        nombreDePommes=10;
    }
}
```

```

public Object ajoutePomme(ChangeRayon t) {
    nombreDePommes=nombreDePommes
        +((Integer)(t.nombreDArticles()).intValue());
    System.out.println("Il y a maintenant "+nombreDePommes+" pomme(s)");
    return new Integer(nombreDePommes);
}

public Object retirePomme(ChangeRayon t) {
    int pommesARetirer = ((Integer)(t.nombreDArticles()).intValue());
    if (nombreDePommes>=pommesARetirer)
        nombreDePommes=nombreDePommes-pommesARetirer;
    else
        pommesARetirer=0;
    System.out.println("Il y reste "+nombreDePommes+" pomme(s) en rayon");
    return new Integer(pommesARetirer);
}

```

On remarque que le constructeur par défaut du rayon fruits et légumes met 10 pommes dans le rayon. Puis les méthodes `ajoutePomme` et `retirePomme` sont implémentées.

Nous allons ensuite rajouter une méthode `main` pour pouvoir créer une instance de `FruitsEtLegumes` :

```

public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager
            (new java.rmi.RMISecurityManager());
    }
    try {
        Rayon monRayon = new FruitsEtLegumes();
        java.rmi.Naming.rebind("maboutique", monRayon);
        System.out.println("FruitsEtLegumes bound");
    } catch (Exception e) {
        System.err.println("FruitsEtLegumes exception: " +
            e.getMessage());
        e.printStackTrace();
    }
}
} // fin de la classe FruitsEtLegumes

```

## 1.5 Le panier

Il reste maintenant à écrire la classe `Panier`. `Panier` qui va demander des pommes au rayon :

```

package panier;

import java.rmi.*;
import boutique.Rayon;
import boutique.ChangeRayon;

public class Panier {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            String name = "/" + args[0] + "/maboutique";
            Rayon fruitsEtLegumes = (Rayon) Naming.lookup(name);
            ChangeRayon changement = new Changement(Integer.parseInt(args[1]));
            Integer pommesObtenues =

```

```

        (Integer)fruitsEtLegumes.retirePomme(changement);
        System.out.println("Il y a "+pommesObtenues
            +" pomme(s) dans le panier.");
    } catch (Exception e) {
        System.err.println("Panier exception: " +
            e.getMessage());
        e.printStackTrace();
    }
}
}
}

```

Il ne reste plus qu'à écrire une implémentation de la classe `boutique.ChangeRayon`, qui va simplement enrober un objet de type `int` :

```

package panier;

public class Changement implements boutique.ChangeRayon {
    int monNombreDArticles=0;
    Changement(int n) {
        monNombreDArticles=n;
    }
    public Object nombreDArticles() {
        return new Integer(monNombreDArticles);
    }
}

```

## 2 Ant

### 2.1 Description de Ant

Ant (<http://ant.apache.org>) est un programme permettant de ne pas taper toutes les informations de compilation sur la ligne de commande. Ant est utilisé par les IDE NetBeans et Eclipse (donc pas besoin de le télécharger si Netbeans ou Eclipse est déjà installé).

Un fichier Ant est un fichier XML, généralement nommé `build.xml`. Ce fichier est l'équivalent d'un Makefile ([www.gnu.org/software/make](http://www.gnu.org/software/make)).

### 2.2 Hiérarchie

On suppose que l'on part de la hiérarchie suivante:

```

| build.xml
| java.policy
+ src
  + boutique
    | | ChangeRayon.java
    | | Rayon.java
  + mesrayons
    | | FruitsEtLegumes.java
  + panier
    | Changement.java
    | Panier.java

```

### 2.3 Compilation avec Ant - le tag `javac`

Un fichier ant commence par une balise `projet` composant la cible par défaut à exécuter (ici `rmic` que nous verrons plus loin):

```
<?xml version="1.0"?>

<project name="jar" basedir="." default="rmic">
  Créons ensuite une cible pour compiler toute notre arborescence située dans le dossier src
  <target name="compile">
    <mkdir dir="build"/>          <!-- creation du repertoire -->
    <javac srcdir="src"           <!-- compilation des *.java -->
      destdir="build"
      debug="on"/>
  </target>
```

Quand cette cible sera exécutée, le répertoire build sera créé, puis toute l'arborescence des fichier \*.java sera copiée dans build en étant transformée en fichier \*.class.

On obtient en plus de la hiérarchie précédente :

```
+ build
+ classes
  + boutique
    | | ChangeRayon.class
    | | Rayon.class
  + mesrayons
    | | FruitsEtLegumes.class
  + panier
    | | Changement.class
    | | Panier.class
```

## 2.4 Exécuter rmic - le tag rmic

Les interfaces distantes doivent être compilées en utilisant un utilitaire spécial nommé rmic.

Il peut être invoqué sur la ligne de commande par la séquence :

```
>rmic -v1.1 -d build/classes mesrayons.FruitsEtLegumes
```

ou de manière similaire par la cible Ant suivante :

```
<target name="rmic" depends="compile">
  <rmic stubversion="1.1"
    classname="mesrayons.FruitsEtLegumes"
    base="build/classes" />
</target>
```

L'attribut depends="compile" sert à demander l'exécution préalable de la cible nommée compile. On obtient en plus de la hiérarchie précédente :

```
+ build
+ classes
  + mesrayons
    | FruitsEtLegumes_Skel.class
    | FruitsEtLegumes_Stub.class
```

Les classes \*\_Skel.class et \*\_Stub.class servent à prendre en charge automatiquement la sérialisation et la communication réseau.

## 2.5 Exécuter le serveur - le tag java

L'exécution du serveur se fait par la ligne de commande suivante :

```
java -Djava.rmi.server.codebase=file:/chemin/vers/les/classes \
-Djava.rmi.server.hostname=localhost \
-Djava.security.policy=java.policy \
-cp build/classes mesrayons.FruitsEtLegumes
```

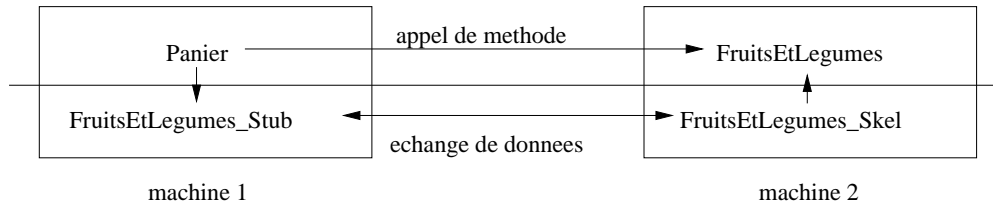


Figure 1: Classes \*\_Skel et \*\_Stub.

ou par la cible Ant suivante:

```

<target name="serveur" depends="rmic">
  <java classname="mesrayons.FruitsEtLegumes" fork="true">
    <sysproperty key="java.rmi.server.codebase" value="file:/chemin/vers/les/classes/" />
    <sysproperty key="java.rmi.server.hostname" value="localhost" />
    <sysproperty key="java.security.policy" value="java.policy" />
    <classpath>
      <!-- rajoute un répertoire -->
      <pathelement path="build/classes" />
      <!-- rajoute une archive jar -->
      <pathelement location="/chemin/vers/archive.jar" />
    </classpath>
  </java>
</target>

```

## 2.6 Effacer les fichiers générés

On peut également rajouter la cible ci-dessous pour effacer les fichiers générés:

```

<target name="clean">
  <delete dir="build" />
  <delete>
    <fileset dir="." includes="**/*~" defaultexcludes="no" />
    <fileset dir="." includes="**/*.jar" defaultexcludes="no" />
  </delete>
</target>

```

Il faut alors fermer la balise `project` pour clore le fichier Ant.

```
</project>
```

## 2.7 Invoquer Ant

Pour générer le jar à partir de tous les fichiers source, il suffit de taper ant

```

>ant
Buildfile: build.xml

compile:
  [mkdir] Created dir: /chemin/vers/build
  [mkdir] Created dir: /chemin/vers/build/classes
  [javac] Compiling 5 source files to /chemin/vers/build/classes

rmic:
  [rmic] RMI Compiling 1 class to /chemin/vers/build/classes

BUILD SUCCESSFUL
Total time: 2 seconds

```

On remarque que c'est tout d'abord la cible `compile` qui s'exécute, puis la cible `rmic`.

Si un fichier source `*.java` a été modifié, seul ce dernier sera compilé par une nouvelle invocation de `ant`. Pour effacer les fichiers générés taper `ant clean`.

## 3 Exécuter les clients et serveurs RMI

### 3.1 `rmiregistry`

Il faut tout d'abord lancer l'entité qui va permettre au serveur de s'enregistrer. Ceci se fait en invoquant sur la ligne de commande :

UNIX (bash):

```
export CLASSPATH=/chemin/vers/mes/point_class
rmiregistry &
```

UNIX (csh):

```
setenv CLASSPATH /chemin/vers/mes/point_class
rmiregistry &
```

Windows:

```
# fixer la variable d'environnement classpath
start rmiregistry
```

Une autre alternative sous Unix ou Windows consiste à lancer `rmiregistry` dans le répertoire où sont les packages contenant les `*.class`.

### 3.2 Lancer le serveur

Une fois le fichier `build.xml` précédent constitué, le serveur se lance simplement par la commande :

```
ant serveur
```

Il faut au préalable avoir créé le fichier `java.policy` dont voici un exemple possible:

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

### 3.3 Lancer le client

Une fois le fichier `build.xml` précédent constitué, le client se lance simplement par la commande :

```
ant client
```

## 4 Exercices

### Exercice 2.1

**Q1** Récupérer le source donné dans le cours à l'url : [http://www.derepas.com/java/cours2\\_exercice\\_1.jar](http://www.derepas.com/java/cours2_exercice_1.jar). Le décompresser à l'aide de la commande `jar xvf cours2_exercice_1.jar`. Mettre les bons chemins dans le fichier `build.xml`. Exécuter le client et le serveur, à l'aide de la commande `ant`.

**Q2** On souhaite rendre l'interface `boutique.Rayon` plus générique. Créer une classe `boutique.Article` qui contient simplement une chaîne de caractères. Transformer les méthodes de `boutique.Rayon` :

```
Object ajoutePomme(ChangeRayon t) throws RemoteException;
Object retirePomme(ChangeRayon t) throws RemoteException;
```

En :

```
Object ajouteArticle(ChangeRayon t) throws RemoteException;
Object retireArticle(ChangeRayon t) throws RemoteException;
```

La classe ChangeRayon étant alors transformée en

```
public interface ChangeRayon extends Serializable {
    Object nombreDArticles();
    /**
     * retourne le type d'article concerné par le changement.
     */
    Object article();
}
```

Transformer les implémentations correspondantes.

## Exercice 2.2

Le but de l'exercice est de créer un mini serveur de chat.

**Q1** Il faut tout d'abord définir les interfaces. On va disposer de deux classes :

- une classe serveur sur laquelle on dispose d'une méthode `ecrit(String nomUtilisateur, String phrase)`, qui permet d'envoyer une phrase pour un utilisateur.
- une classe client sur laquelle on dispose d'une méthode `notifie(String nomUtilisateur, String phrase)`, qui permet à un client de chat de recevoir du texte.

Écrire ces interfaces.

**Q2** Écrire une implémentation des interfaces précédentes permettant de réaliser le serveur de chat.