Etudier le fonctionnement utilisateur de rmi

I Fonctionnement de base

L'objectif de ce premier exercice est de faire un premier service distant réalisant une fonction simple comme le calcul de la suite de fibonacci. Le code d'exemple est le suivant :

```
package service ;
public class Fibo {

  public int calcul(int val){
    int val1=0,val2=1,val3=0;
    for (int i=0; i<val; i++){
     val3=val1+val2;
     val1=val2;
     val2=val3;
     System.out.println("val "+val3);
    }
    return val3;
  }

  public static void main (String [] arg) throws Exception{
    Fibo s=new Fibo();
    System.out.println("Calcul
"+s.calcul(Integer.parseInt(arg[0])));
  }
}
```

I.1 Réaliser cette classe, compiler tester.

*Remarque : fabriquez vous un environnement de travail « propre » contenant les répertoires suivants : src et classes

*Remarque : utilisez l'option –d de javac permettant de déposer le résultat dans un répertoire différent des sources.

*Remarque : la classe Fibo s'appelle en fait service.Fibo et doit donc se trouver dans ~src/service/Fibo.java et le résultat de la compilation doit se trouver dans ~/classes/service/Fibo.class

I.2 Réaliser l'interface, une implementation et un client

Afin de rendre le code plus « modulaire », il est intéressant de « refactorer » le code afin d'avoir une indépendance entre l'interface et l'implémentation d'un service. Modifier votre code afin d'avoir une interface de service **service.Fibo**, un classe d'implémentation **service.FiboImpl**, un client qui utilise le service (déplacer et modifier la méthode main dans une classe **client.Client**). Modifier le code client afin que la partie gauche d'une instanciation soit une interface et la partie droite une implémentation.

Vous devez alors avoir 3 classes de base : service.Fibo, service.FiboImpl, client.Client

I.3 Fabriquer le service distant

Modifier l'interface de service afin de la rendre *compatible* rmi. C'est à dire elle doit étendre l'interface **java.rmi.Remote** et chaque méthode doit pouvoir lever une **java.rmi.RemoteException.** Préparer l'interface, compiler.

Le serveur quant à lui doit étendre la classe **java.rmi.server.UnicastRemoteObject** et implanter votre interface de service. Compiler. D'autre part, il est nécessaire de rendre le serveur disponible aux clients (mise à disposition du stub de manipulation). Pour cela, il faut qu'une fonction du système fabrique l'instance du serveur et dépose le stub dans un annuaire. Cette étape est réalisée par le code suivant :

```
public static void main(String [] arg) throws Exception{
  service.Fibo srv=new service.FiboImpl();
  java.rmi.Naming.bind(HelloIfc.class.getName(), srv);
}
```

Remarque : HelloIfc.class.getName(), permet de renvoyer une chaîne de caractères représentant le nom de la classe. On aurait pu choisir n'importe quel autre chaîne de caractères.

Compiler – Executer… Que se passe t'il ? identifier clairement l'erreur concernée.

I.4 Fabrication du stub et du skeleton.

Pour fabriquer le stub/skeleton il faut utiliser le compilateur **rmic** qu'il faut appliquer sur la classe du serveur (oui la classe).
Remarque : n'oubliez pas de générer les stubs dans le répertoire classes (option –d)

I.5 Lancer l'annuaire

Aller dans le répertoire classes, puis lancer le programme rmiregistry. (C'est le service d'annuaire). Sur quel port tourne le service d'annuaire ?
Lancer le server, normalement le stub de manipulation du service est déposé (bind, association nom/objet) sur l'annuaire).

I.6 Lancer le client

Dans le répertoire client modifier le code afin qu'il récupère le stub sur l'annuaire. L'appel devient :

```
HelloIfc hello=(HelloIfc)Naming.lookup(
     "rmi://ares-frenot-3/"+HelloIfc.class.getName());
```

Compiler exécuter….

II Séparation des classes

Créer trois répertoires client / serveur / registry, dans lesquelles seules les sources sont installées (source du client, source du serveur, l'annuaire ne dépend d'aucune source).
Puis installer ou compiler les classes concernées.
Quelles sont les classes nécessaires aux différents éléments.

III Chargement dynamique des stubs

Il est possible de ne pas copier les classes des stubs sur le client. Celui-ci récupère les classes à partir d'une url. Pour cela il faut lancer la machine virtuelle du client avec un paramètre indiquant l'url à partir de laquelle il peut charger les classes qui lui manque.

```
java -Djava.rmi.server.codebase=http://serverhost/~username/rmi/ client.Client
ou
java - Djava.rmi.server.codebase=file:///home/sfrenot/tmp/service/classes
clietn.Client
```

IV Système distribué

Utiliser votre client pour utiliser le service provenant d'une machine d'un autre binôme. Vérifier que votre code fonctionne sur les serveur des autres groupes.

V Objet local et objet remote

Modifier votre code afin de récupérer un objet contenant toutes les valeurs intermédiaires du calcul de la suite de fibonacci. Tester les deux modes de fonctionnement : par sérialisation de l'objet contenant les valeurs de la suite et par fabrication d'un stub de manipulation distant.

VI Pour les tenaces

Modifiez le code afin que le client fournisse un objet au serveur distant. L'objet fournit par le client peu bien évidemment être local (donc sérialisé) ou distant (le client fabrique un stub de manipulation au serveur).
Et la boucle est bouclée…

V Remarques de fin

- Étudier les sources générées par le compilateur rmid (option –keepgenerated)
- Exploiter l'annuaire afin de déposer des instances qui ne sont pas liées à des Stubs (exemple carnet d'adresse)