Polytech Marseille

TD 2
Java RMI

Informatique - Formation par alternance HUGo

Exercice 1. Réservation d'hôtels

On considère une chaîne hôtelière offrant un service de réservation accessible à distance via Java RMI. Le service gère plusieurs hôtels. Chaque hôtel dispose d'un certain nombre de chambres pouvant être réservées. Les données concernant chaque hôtel sont stockées indépendamment les unes des autres sur le serveur.

Le service offre trois opérations : `reserver`, `annuler` et `lister`. Cette dernière (`lister`) retourne, à partir d'un numéro de réservation, un objet de type `Infos` qui contient les caractéristiques de la réservation correspondante (nom du client, nom de l'hôtel, date d'arrivée, nombre de nuits, nombre de chambres).

1. Donnez l'interface Java RMI définissant les trois opérations spécifiées ci-dessus.
2. Ecrivez la déclaration de la classe `Infos`.
3. Evoquez une ou plusieurs exceptions pouvant être définies. Que faut-il modifier dans le code ?
4. Est-ce que l'objet serveur RMI, jouant le rôle de la chaîne hôtelière, peut exécuter simultanément plusieurs fois la méthode `reserver` ? Si oui, que faut-il faire ? Si non, pourquoi ?

Exercice 2. Calendrier partagé

Vous êtes chargés de réaliser un service (simpliste) de calendrier partagé en utilisant Java RMI. Le service est composé d'un serveur central sur lequel un objet `CalServeur` est invocable à distance. Chaque calendrier d'utilisateur (créé par un appel distant de la méthode `createCal` de `CalServeur`) est représenté par un objet de type `Cal` (placé sur le serveur). Pour simplifier l'implantation, on ne représente pas des rendez-vous heure par heure, mais simplement en indiquant si les gens sont libres un jour donné. Un jour sera représenté par un numéro. Un utilisateur pourra indiquer dans son calendrier si il est occupé tel jour, libérer une journée, ...

L'interface `CalServeurInterface` est composée des méthodes suivantes :
- `createCal` : prend un nom d'utilisateur en paramètre et retourne un objet de type `Cal`.
- `intersectionLibre` : prend en paramètre un tableau de noms d'utilisateurs et un jour, et retourne un booléen indiquant si le groupe d'utilisateurs est libre le jour spécifié.
- `reserve` : prend en paramètre un tableau de noms d'utilisateurs et un jour, et ne retourne rien. Cette méthode pose une réunion au groupe d'utilisateurs le jour spécifié.

L'interface `CalInterface` est composée des méthodes suivantes :
- `estLibre` : prend en paramètre un jour, et retourne un booléen indiquant si l'utilisateur est disponible le jour donné.
- `libere` : prend en paramètre un jour, et retourne un booléen indiquant si la libération du jour donné a bien été effectuée.
- `occupe` : prend en paramètre un jour, et retourne un booléen indiquant si l'occupation du jour donné a bien été effectuée.

1. Quel est le concept mis en oeuvre par la méthode `createCal` de la classe `CalServeur` ?
2. Donnez les interfaces Java RMI. Pensez à définir d'éventuelles exceptions qui pourraient être levées, et à les prendre en compte dans les interfaces.
3. Donnez une ébauche rapide de la classe `CalServeur`. Vous êtes libre d'ajouter des attributs et des méthodes que vous jugez nécessaires.
4. Dans la version actuelle, il existe un problème qui se révélera lors de l'utilisation. Quel est-il ? Quel est l'origine ? Donnez un petit scénario d'exécution illustrant ce problème. Que proposez-vous pour y remédier ?

Exercice 3. Réveil et "appel en retour"

On souhaite programmer un service de réveil RMI basé sur un mécanisme de *callback*. Le principe est le suivant :

- le client fabrique un objet `CallBack` distant ;
- le client enregistre l'objet auprès d'un serveur de réveil,
- le client s'endort sur l'objet enregistré (grâce au mécanisme de `wait` des moniteurs) ;
- quand le serveur souhaite réveiller le client, il effectue un appel distant sur `CallBack` (le client est réveillé grâce à `notify`).

1. Proposez une interface `CallBackInterface` permettant au serveur de réveiller son client.
2. Proposez une interface `ReveilInterface` représentant le serveur.
3. Proposez une classe `CallBack` implémentant `CallBackInterface` réalisant les fonctionnalités souhaitées.
4. Proposez une classe représentant un futur objet serveur (`Reveil`) qui réveille le client après 10 secondes d'attente.
   Attention : un appel à `wait` (moniteur) doit provoquer une attente du client, et non l'appel à la méthode distante.

Exercice 4. Tchat (version centralisée, en Java RMI)

L'objectif est de concevoir une application de Tchat basée sur un intergiciel orienté objet, ici Java RMI. Une version simple consiste à avoir un serveur qui reçoit chaque message et le fait suivre aux autres clients. On obtient donc une architecture centralisée ("en étoile", avec le serveur au centre).

Dans cette version, le serveur de Tchat enregistre et désenregistre des clients distants. C'est aussi lui qui fait suivre les messages envoyés par les clients à tous les autres clients.

En ce qui concerne un client, il peut s'enregistrer et se désenregistrer du serveur de Tchat. Lors de son enregistrement, il indique son pseudo (nom de l'utilisateur). Il peut afficher un message reçu et demander au serveur de Tchat d'envoyer un message à tout le monde.

Remarque : n'utilisez pas de callback (pas d'appels en retour).

ANNEXE : Exemple "Hello World" en Java RMI

Objectif : lire, comprendre, réaliser et tester l'exemple "Hello World".

1. Définition de l'interface (HelloInterface.java)

```
package exemples.javarmi.hello;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HelloInterface extends Remote {
  /**
   * méthode imprimant un message prédéfini dans l'objet appelé
   */
  public String sayHello() throws RemoteException ;
}
```

2. Définition de la classe réalisant l'interface (HelloImpl.java)

```
package exemples.javarmi.hello;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject implements HelloInterface {
  private static final long serialVersionUID = 6586708515447619453L;
  private String message;

  public HelloImpl (String s) throws RemoteException {
        message = s;
  }

  public String sayHello () throws RemoteException {
        return message ;
  }
}
```

3. Ecriture du programme du serveur (HelloServer.java)

```
package exemples.javarmi.hello;
import java.rmi.Naming;
import java.rmi.RMISecurityManager;

public class HelloServer {
  public static void main (String[ ] args) {
        /* lancer SecurityManager */
        //System.setSecurityManager (new RMISecurityManager ());  // deprecated
        try {
                /* créer une instance de la classe Hello et
                        l'enregistrer dans le serveur de noms */
                Naming.rebind ("Hello1", new HelloImpl ("Hello world"));
                System.out.println ("Serveur prêt") ;
        }
        catch (Exception e) {
                System.out.println("Erreur serveur : " + e) ;
        }
  }
}
```

4. Ecriture du programme du client (HelloClient.java)

```
package exemples.javarmi.hello;
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
```

```
public class HelloClient {
  public static void main (String[ ] args){
        /* lance le SecurityManager */
        //System.setSecurityManager (new RMISecurityManager ()); // deprecated
        try {
                String server = args[0];
                int port = Integer.parseInt(args[1]);
                /* cherche référence objet distant */
                HelloInterface hello = (HelloInterface) Naming.lookup(
                                "rmi://"+server+":"+port+"/Hello1");
                /* appel de méthode à distance */
                System.out.println (hello.sayHello()) ;
        }
        catch (Exception e) {
                System.out.println ("Erreur client : " + e) ;
        }
  }
}
```

5. Définition de la politique de sécurité (fichier "security.policy")

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

ou (mais dangereux...)

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

6. Compilation

Il faut compiler toutes les classes :

```
javac exemples/javarmi/hello/*.java
```

Il faut créer les talons client et serveur :

```
rmic exemples.javarmi.hello.HelloImpl
```

Cela génère Hello_Stub.class (talon client) et Hello_Skel.class (talon serveur).

8. Déploiement et exécution

On suppose que l'élève "e00000" a placé le dossier "exemples" dans un dossier "SR".
On suppose que le serveur tourne sur la machine 134.192.30.6 et le registre utilise le port par défaut 1099 (le registre rmi tourne aussi sur la machine 134.192.30.6).

Sur la machine "serveur" (laquelle connaît toutes les classes, sauf HelloClient) :

Lancement du registre rmi :

```
rmiregistry
```

Se placer dans le dossier "SR".

Lancement du programme serveur (commande sur une seule et même ligne) :

```
java -cp .
-Djava.security.policy=/filer/eleves/e00000/SR/exemples/security.policy
-Djava.rmi.server.codebase=file:/filer/eleves/e00000/SR/
  exemples.javarmi.hello.HelloServer
```

Sur la machine "client" (laquelle connaît seulement l'interface au sens RMI, le programme client et la souche (stub)) :

Lancement du programme client (commande sur une seule et même ligne) :

```
java -cp .
-Djava.security.policy=/filer/eleves/e00000/SR/exemples/security.policy
-Djava.rmi.server.codebase=file:/filer/eleves/e00000/SR/
  exemples.javarmi.hello.HelloClient 134.192.30.6 1099
```

Remarque importante : pour une utilisation sur plusieurs machines, vous devez (en plus) utiliser les options suivantes :

```
-Djava.rmi.server.hostname=adresseIPserveur
-Djava.rmi.server.disableHttp=true
```

Voir la documentation officielle pour plus d'information.