Les exercices de cet examen portent tous sur le système RMI de `Java`. La consultation de documents (supports de cours, livres, etc.) est autorisée. L'interaction avec les autres candidats n'est pas autorisée !

Exercice 1 :

On souhaite définir un objet distant RMI proposant la méthode distante de signature suivante :

```
1  B truc(A a)
```

Questions :

1. Ecrivez l'interface `IDistant` correspondante en supposant données les classes ou interfaces `A` et `B`.
2. On souhaite que le paramètre de la méthode soit transmis par valeur du client au serveur. Quelle(s) condition(s) doit (doivent) être satisfaite(s) par `A` ?
3. On souhaite que le résultat de la méthode corresponde à un objet distant. Quelle(s) condition(s) doit (doivent) être satisfaite(s) par `B` ?

Exercice 2 :

On suppose donnée la classe suivante :

Entier

```
1  import java.io.Serializable;
2  public class Entier implements Serializable {
3    public int n;
4    public Entier(int p) {
5      n=p;
6    }
7  }
```

On considère le programme suivant :

Local

```
1  public class Local {
2    public void inc(Entier x) {
3      x.n++;
4    }
5    public static void main(String[] args) {
6      Local l=new Local();
7      Entier y=new Entier(2);
8      System.out.println(y.n);
9      l.inc(y);
10     System.out.println(y.n);
11   }
12 }
```

Questions :

1. Quel est l'affichage produit par le programme ?
2. On suppose donnée une interface distante `RLocal` proposant une méthode `inc` de même signature que celle de l'objet `Local`, c'est-à-dire prenant comme paramètre un objet `Entier` et ne renvoyant rien. Soit donc :

RLocal

```
1  import java.rmi.Remote;
2  import java.rmi.RemoteException;
3  public interface RLocal extends Remote {
4    public void inc(Entier x) throws RemoteException;
5  }
```

Ecrivez l'équivalent du `main` de la classe `Local`, avec comme seule différence l'utilisation de l'objet distant d'interface `RLocal` (on supposera qu'une implantation de cette interface est stockée dans le `rmiregistry` avec comme nom `//localhost/ILocal`).

3. Peut-on déterminer complètement l'affichage du `main` réalisé dans la question précédente sans connaître la programmation de l'objet distant ? Si oui, quel est cet affichage et pourquoi peut-on le déterminer ?

Exercice 3 :

On propose l'interface distante suivante :

ITableau

```
1  import java.rmi.Remote;
2  import java.rmi.RemoteException;
3  public interface ITableau extends Remote {
4    public double moyenne(double[] x) throws RemoteException;
5  }
```

Comme son nom l'indique, la méthode `moyenne` renvoie la moyenne du tableau qu'elle reçoit en paramètre.

Questions :

1. Ajoutez à l'interface `ITableau` une méthode `min` qui renvoie la **position** du plus petit élément du tableau qu'elle reçoit en paramètre.
2. Ecrivez une implantation `Tableau` de l'interface `ITableau`.
3. Ecrivez un programme qui crée un objet `Tableau` et le rend disponible par le `rmiregistry` sous le nom `//localhost/Moyenne`.
4. Donnez le diagramme de classes du système, en incluant les éventuelles classes engendrées par `rmic`.
5. On souhaite développer le client sans utiliser le mécanisme de déploiement automatique :
   (a) quels fichiers `class` doivent figurer dans le `CLASSPATH` du compilateur pour le développement du client ?
   (b) quels fichiers `class` doivent figurer dans le `CLASSPATH` du `rmiregistry` ?
   (c) quels fichiers `class` doivent figurer dans le `CLASSPATH` de la JVM[1] pour le démarrage du client ?

Exercice 4 :

On propose l'interface distante suivante :

ICompteur

```
1  import java.rmi.Remote;
2  import java.rmi.RemoteException;
3  public interface ICompteur extends Remote {
4    public int next() throws RemoteException;
5  }
```

[1] JVM : *Java Virtual Machine*, Machine Virtuelle Java

Le principe de cet objet distant est de renvoyer une nouvelle valeur à chaque appel de `next`, sachant que la valeur initiale est zéro et qu'un appel de `next` ajoute un à cette valeur avant de renvoyer la nouvelle valeur. On doit donc obtenir 1 au premier appel, 2 au deuxième, etc. On propose la programmation suivante de cette interface :

Compteur

```
1  import java.rmi.Remote;
2  import java.rmi.RemoteException;
3  import java.rmi.server.UnicastRemoteObject;
4  public class Compteur extends UnicastRemoteObject implements ICompteur {
5    private int val;
6    public Compteur() throws RemoteException {
7      val=0;
8    }
9    public int next() throws RemoteException {
10     int oldval=val;
11     try {
12       // pause de 1000 ms (i.e., 1 seconde)
13       Thread.sleep(1000);
14     } catch (InterruptedException e) {};
15     oldval++;
16     val=oldval;
17     return val;
18   }
19 }
```

Le scénario d'utilisation est le suivant : deux clients (correspondant à 2 JVM distinctes) appellent "simultanément" la méthode `next` du serveur (en fait, le client A se connecte, puis 100 ms plus tard, le client B se connecte).

Questions :

1. Combien de temps (approximativement) chaque client attend-il la réponse du serveur ?
2. Quelle est la réponse renvoyée à chaque client ?
3. Comment obtenir simplement une réponse plus cohérente (en modifiant l'implantation du serveur) ? Quelle est alors le temps d'attente de chaque client, toujours pour le même scénario, mais avec le nouveau serveur ?

Exercice 5 :

On propose l'interface et les classes suivantes :

One

```
1  import java.io.Serializable;
2  public class One implements Serializable {
3    public void one() {
4      System.out.println("One.one()");
5    }
6  }
```

ITwo

```
1  import java.rmi.Remote;
2  import java.rmi.RemoteException;
3  public interface ITwo extends Remote {
```

```
4   public void two() throws RemoteException;
5 }
```

Two

```
1 import java.rmi.Remote;
2 import java.rmi.RemoteException;
3 import java.rmi.server.UnicastRemoteObject;
4 public class Two extends UnicastRemoteObject implements ITwo {
5   public Two() throws RemoteException { }
6   public void two() throws RemoteException {
7     System.out.println("Two.two()");
8   }
9 }
```

Ces éléments sont utilisés pour la communication entre un client et un serveur. L'interface du serveur est la suivante :

IServeur

```
1 import java.rmi.Remote;
2 import java.rmi.RemoteException;
3 public interface IServeur extends Remote {
4   public void a(One o) throws RemoteException;
5   public One b() throws RemoteException;
6   public void c(ITwo o) throws RemoteException;
7   public ITwo d() throws RemoteException;
8 }
```

On cherche à savoir quelle JVM va exécuter les différentes méthodes du système (JVM client ou JVM serveur). On propose la programmation suivante pour le serveur :

Serveur

```
1 import java.rmi.Remote;
2 import java.rmi.RemoteException;
3 import java.rmi.server.UnicastRemoteObject;
4 public class Serveur extends UnicastRemoteObject implements IServeur {
5   public Serveur() throws RemoteException { }
6   public void a(One o) throws RemoteException {
7     o.one();
8   }
9   public One b() throws RemoteException {
10    return new One();
11  }
12  public void c(ITwo o) throws RemoteException {
13    o.two();
14  }
15  public ITwo d() throws RemoteException {
16    return new Two();
17  }
18 }
```

Questions :

1. Le client crée un objet `One` et appelle la méthode `a` avec cet objet comme paramètre : par quelle JVM cette méthode est-elle exécutée ?

2. L'implantation de la méthode `a` appelle la méthode `one` grâce à son paramètre `o` : par quelle JVM la méthode `one` est-elle exécutée ?
3. Le client appelle la méthode `b`, puis la méthode `one` sur le résultat de `b` (i.e., un appel de la forme `s.b().one()`, où `s` désigne une variable de type `IServeur` contenant une référence vers l'objet serveur distant) : par quelle JVM la méthode `one` est-elle exécutée ?
4. L'implantation de la méthode `c` appelle la méthode `two` grâce à son paramètre `o` : par quelle JVM la méthode `two` est-elle exécutée ?
5. Le client appelle la méthode `d`, puis la méthode `two` sur le résultat de `d` (i.e., un appel de la forme `s.d().two()`, où `s` désigne une variable de type `IServeur` contenant une référence vers l'objet serveur distant) : par quelle JVM la méthode `two` est-elle exécutée ?
6. Quand on démarre le serveur et le client, les JVM correspondantes créent chacune un `Thread` correspondant à la méthode `main`. Pour les quatre cas considérés (questions 2 à 5), indiquez si la méthode étudiée est exécutée par le `Thread` principal ou par un autre `Thread`.
7. Que se passe-t-il si dans l'interface `IServeur` (et dans la classe `Serveur`), on remplace toutes les occurrences de `ITwo` par `Two` ?

Exercice 6 :

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

- le client fabrique un objet `CallBack` distant ;
- le client enregistre l'objet auprès d'un serveur de réveil, avec comme paramètre un objet `Condition` qui décrit les conditions de réveil du client ;
- le client s'endort sur l'objet enregistré (grâce au mécanisme de `wait`) ;
- quand le serveur souhaite réveiller le client, il effectue un appel distant sur `CallBack` (le client est réveillé grâce à `notify`).

On suppose donnée une interface `Condition` (vide dans un premier temps).

Questions :

1. Proposez une interface `CallBack` permettant au serveur de réveiller son client.
2. Proposez une interface `Reveil` représentant le serveur.
3. Proposez une programmation de `CallBack` (`CallBackImpl`) réalisant les fonctionnalités souhaitées.
4. Proposez une programmation du serveur (`ReveilImpl`) qui réveille le client après 10 secondes d'attente (sans tenir compte de la `Condition`). Attention, il faut impérativement que l'enregistrement du client soit non bloquant. C'est le mécanisme de `wait` qui provoque une pause du client, pas l'appel à la méthode du serveur.