# Système Distribué avec RMI 2

Yassamine Seladji

yassamine.seladji@gmail.com

19 avril 2018

# Introductions

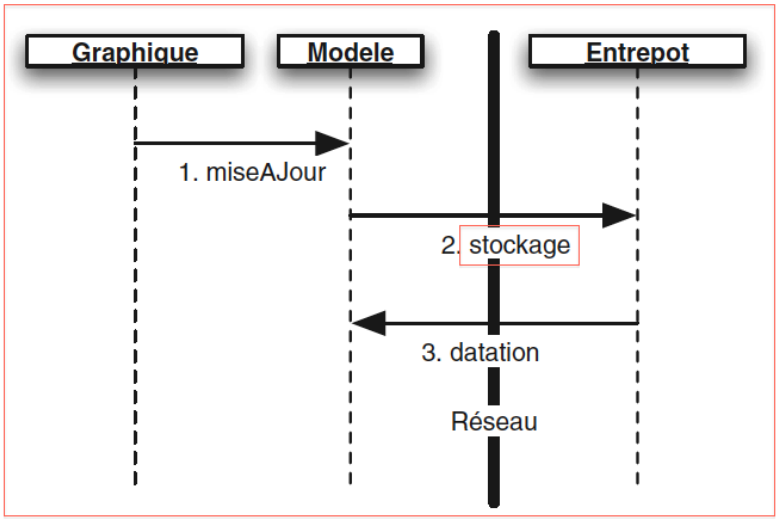
# Passage des arguments

- Passage des arguments par **références**.

# Passage des arguments

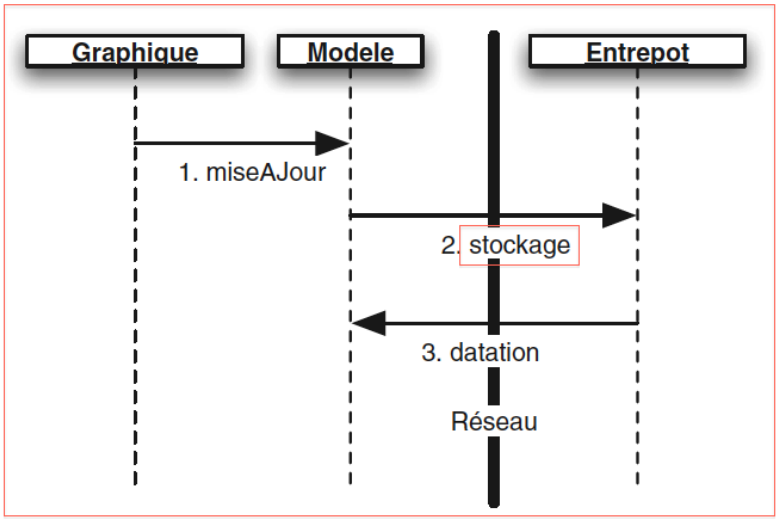
- Passage des arguments par **références**.
- Passage des arguments par **valeurs**.

# Passage des arguments par références



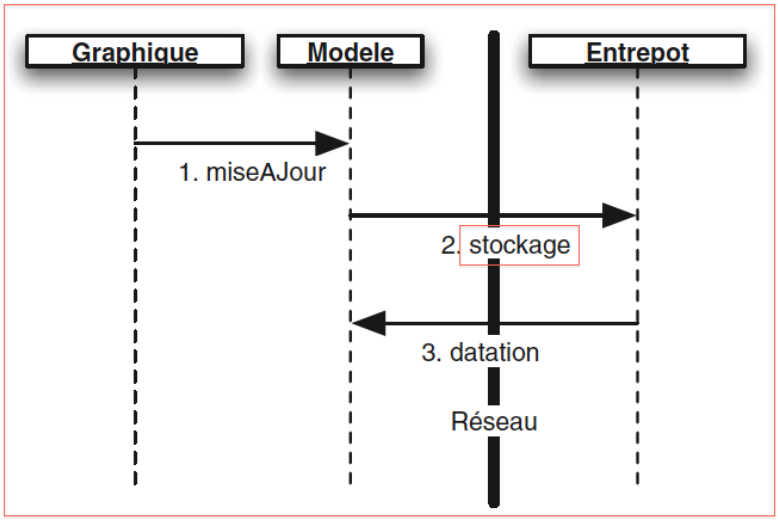

- La méthode *stockage* est une méthode de la classe **Entrepot**.

# Passage des arguments par références




- La méthode *stockage* est une méthode de la classe **Entrepot**.
- La méthode *stockage* prend en paramètre un objet de la classe **Modele**, qui représente un objet **distant**.

# Passage des arguments par références




- La méthode *stockage* est une méthode de la classe **Entrepot**.
- La méthode *stockage* prend en paramètre un objet de la classe **Modele**, qui représente un objet **distant**.
- L'objet distant représente lui même une référence.

# Passage des arguments par références

## L'implémentation de l'entrepôt

- L'interface de l'entrepôt :

```
public interface EntrepotInterface extends Remote{
   public void stockage(ModelInterface modele) throws RemoteException;
}
```

- L'implémentation de l'entrepôt :

```
public class Entrepot extends UnicastRemoteObject implements EntrepotInterface{

   public Entrepot () throws RemoteException{
       super();
   }
   public void stockage(ModelInterface modele) throws RemoteException {
       Date estampille = (Date) modele.datation();
       System.out.println("Modèle stocké le "+estampille.toString());

   }
```

# Passage des arguments par références

## L'implémentation de l'entrepôt

- La mise en place du serveur de l'entrepôt :

```
public class Serveur {
   public static void main(String[] args) {
        Entrepot entrepot;
        try {
            entrepot = new Entrepot();
            LocateRegistry.createRegistry(1098);
            Naming.bind("rmi://localhost:1098/entrepot", entrepot);

        } catch (RemoteException e) {
            System.out.println("Erreur de connexion au serveur entrepôt");
            e.printStackTrace();
        }catch (MalformedURLException e) {
            System.out.println("Erreur sur le nom logique de l'objet distant");
            e.printStackTrace();
        } catch (AlreadyBoundException e) {
            System.out.println("La référence de l'objet distant existe déja");
            e.printStackTrace();
        }

        System.out.println("serveur entrepot connecté");

   }
```

# Passage des arguments par références

## L'implémentation du Modèle

- Définir l'interface distante du Modèle
- Définir la classe **Model** qui implémente l'interface distante.
- La classe **Model** hérite d'**UnicastRemoteObject**.
- Les méthode de la classe **Model** doivent lever une exception de type **RemoteException**.

# Passage des arguments par références

## L'implémentation du Modèle

- L'interface du Modèle :

```
public interface ModelInterface extends java.rmi.Remote{
    public Date datation() throws java.rmi.RemoteException, InterruptedException;
}
```

- L'implémentation du Modèle :

```
public class Model extends UnicastRemoteObject implements ModelInterface {
    private Date estampille;
    public Model() throws RemoteException {
        super();
        this.estampille = new GregorianCalendar().getTime();
    }
    public Date datation() throws RemoteException, InterruptedException {
        Thread.sleep(2000);
        return this.estampille;
    }
    public void miseAjour(){}
    public Date getEstampille() {
        return estampille;
    }
    public void setEstampille(Date estampille) {
        this.estampille = estampille;
    }
```

# Passage des arguments par références

## L'implémentation du Modèle

- La mise en place du client de l'entrepôt :

# Passage des arguments par références

## L'implémentation du Modèle

- La mise en place du client de l'entrepôt :
- Le modèle obtient une référence sur l'entrepôt.

# Passage des arguments par références

## L'implémentation du Modèle

- La mise en place du client de l'entrepôt :
- Le modèle obtient une référence sur l'entrepôt.
- L'entrepôt rappelle le modèle de manière simple.

# Passage des arguments par références

## L'implémentation du Modèle

- La mise en place du client de l'entrepôt :
- Le modèle obtient une référence sur l'entrepôt.
- L'entrepôt rappelle le modèle de manière simple.

```
public static void main(String[] args) {
    Model modele;
    try {
        modele = new Model();
        EntrepotInterface entrepot = (EntrepotInterface)
                Naming.lookup("rmi://localhost:1098/entrepot");
        entrepot.stockage(modele);
    }catch (RemoteException e) {
        System.out.println("Erreur de connexion au serveur entrepôt");
        e.printStackTrace();
    }
```

# Passage des arguments par références

## L'égalité des références

- Deux appels successifs à la même référence.

# Passage des arguments par références

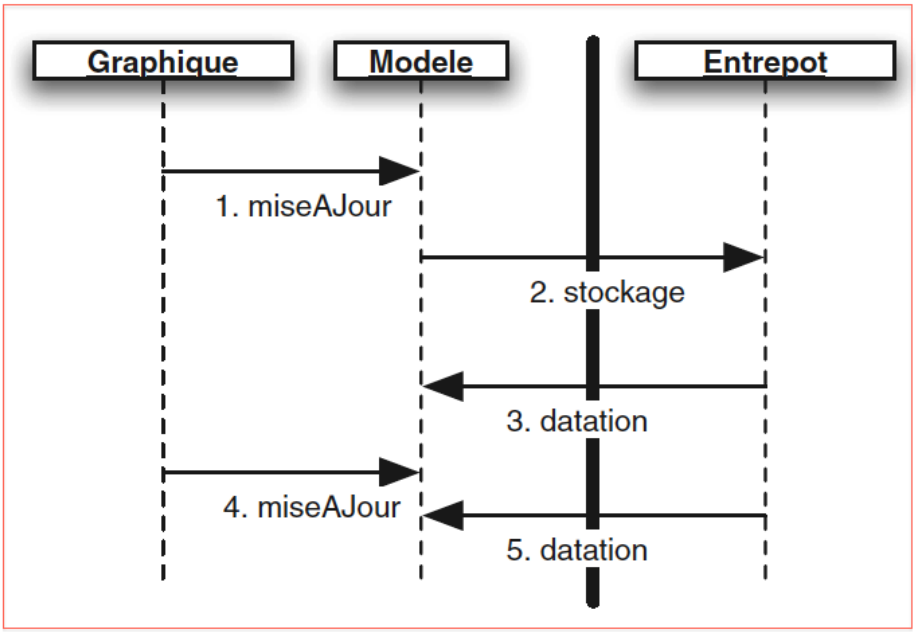
## L'égalité des références

- Deux appels successifs à la même référence.
- Pour cela : ajoutant un attribut **estampille** à l'objet **modèle**.

# Passage des arguments par références

## L'égalité des références

- Deux appels successifs à la même référence.
- Pour cela : ajoutant un attribut **estampille** à l'objet **modèle**.



# Passage des arguments par références

## L'égalité des références

- Si le modèle change de date (d'estampille).

# Passage des arguments par références

## L'égalité des références

- Si le modèle change de date (d'estampille).
- Alors l'appel de la méthode **datation** par l'**entrepôt** reflètera se changement.

| Côté modèle | Côté entrepôt |
|---|---|
| `setEstampille(valeur1)` | `modele.datation()` renvoie `valeur1` |
| `setEstampille(valeur2)` | `modele.datation()` renvoie `valeur2` |

# Passage des arguments

- Passage des arguments par **références**.

# Passage des arguments

- Passage des arguments par **références**.
- Passage des arguments par **valeurs**.

# Passage des arguments par valeurs

Passer l'objet **modèle** par valeur :

- Modèle devient un objet simple.

# Passage des arguments par valeurs

Passer l'objet **modèle** par valeur :

- Modèle devient un objet simple.
- La classe **Modèle** n'hérite plus de **UnicastRemoteObject**.

# Passage des arguments par valeurs

Passer l'objet **modèle** par valeur :

- Modèle devient un objet simple.
- La classe **Modèle** n'hérite plus de **UnicastRemoteObject**.
- La classe Modèle implémente **Serializable**.

```
public class Modele implements Serializable{

    private Date estampille;
    public Modele() {
        this.estampille = new GregorianCalendar().getTime();
    }
    public Date datation() {
        return this.estampille;
    }
    public void setEstampille(Date estampille) {
        this.estampille = estampille;
    }
```

# Passage des arguments par valeurs

Passage de l'objet **modèle** par valeur :

- L'objet **modèle** passé en paramètre de la méthode **stockage** est passé par valeur.

# Passage des arguments par valeurs

Passage de l'objet **modèle** par valeur :

- L'objet **modèle** passé en paramètre de la méthode **stockage** est passé par valeur.
- L'entrepôt utilise l'objet modèle de la même manière que son appel par référence.

# Passage des arguments par valeurs

Passage de l'objet **modèle** par valeur :

- L'objet **modèle** passé en paramètre de la méthode **stockage** est passé par valeur.
- L'entrepôt utilise l'objet modèle de la même manière que son appel par référence.
- Le serveur de l'entrepôt contient la classe de l'objet modèle en local.

# Passage des arguments par valeurs

Passage de l'objet **modèle** par valeur :

- L'objet **modèle** passé en paramètre de la méthode **stockage** est passé par valeur.
- L'entrepôt utilise l'objet modèle de la même manière que son appel par référence.
- Le serveur de l'entrepôt contient la classe de l'objet modèle en local.
- La méthode **datation** sera appelée en local sur une copie du modèle.

```
public void stockage(ModelInterface modele) throws RemoteException {
    Date estampille = (Date) modele.datation();
    System.out.println("Modèle stocké le "+estampille.toString());

}
```

# Passage des arguments par valeurs

Passage de l'objet **modèle** par valeur :

- L'entrepôt doit disposer de l'implémentation de la classe **Modèle**.

# Passage des arguments par valeurs

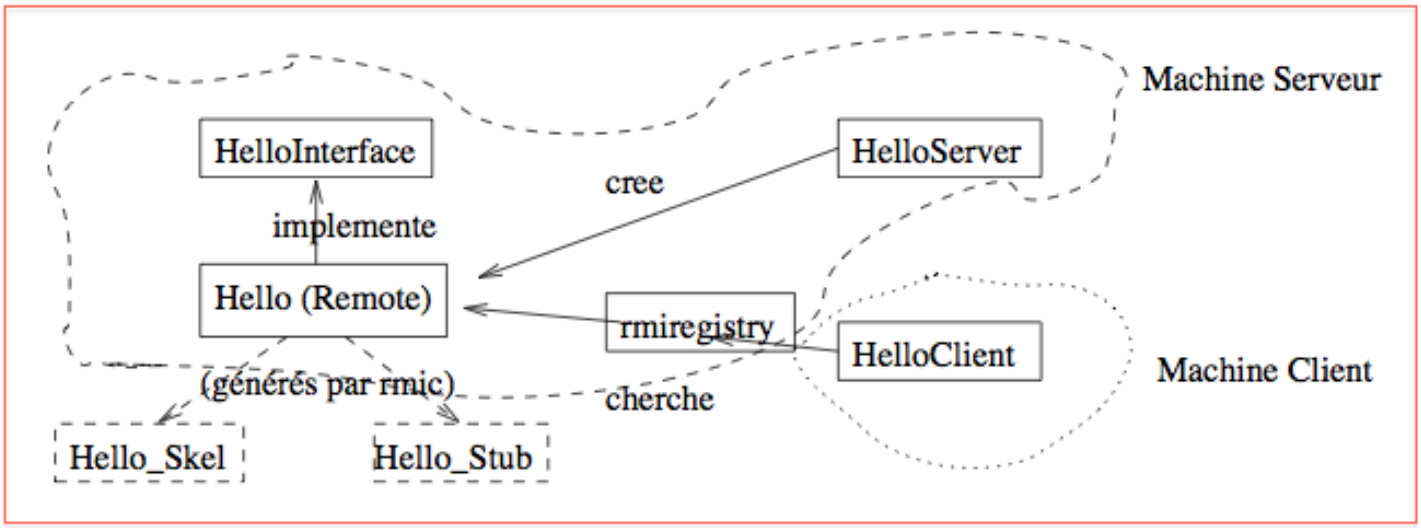
Passage de l'objet **modèle** par valeur :

- L'entrepôt doit disposer de l'implémentation de la classe **Modèle**.
- Si l'objet modèle change d'estampille, sa copie dans l'entrepôt ne sera pas modifier.

| Côté modèle | Côté entrepôt |
|---|---|
| `setEstampille(valeur1)` | `modele.datation()` renvoie `valeur1` |
| `setEstampille(valeur2)` | `modele.datation()` renvoie toujours `valeur1` |

# Passage des arguments : Exercice 1

Reprenant l'exercice de l'application répartie "Hello World" présentée dans le cours précédent.



# Passage des arguments : Exercice 1

Proposer une nouvelle version de l'application tel que :

- le client est représenté par son ID et son nom.
- A la demande du client, le serveur envoie le message "Hello World" en plus du nom du client.

Quels sont les changement à faire dans l'ancienne implémentation.

# Passage des arguments : Correction

- Implémenter la classe **Client** qui hérite de **Serializable**, côté client et serveur.

```
public class Client implements Serializable{
    private int id;
    private String nom;
    public Client(int id,String nom){
        this.nom = nom;
        this.id = id;
        }

    public int getId() {
        return id;
    }
    public String getNom() {
        return nom;
    }
```

# Passage des arguments : Correction

- Le côté client :

```
public class HelloClient {
   public static void main(String[] argv) {
       try {
           HelloInterface hello = (HelloInterface) Naming.lookup
                   ("rmi://localhost:1099/Hello");
           System.out.println(hello.say(new Client(0,"premier")));
       } catch(Exception e) {
           System.out.println("HelloClient exception: "+e);
       }
   }
```

# Passage des arguments : Correction

- Côté serveur :

```
public interface HelloInterface extends Remote {
    public String say(Client c) throws RemoteException;
}
```

```
public class Hello extends UnicastRemoteObject implements HelloInterface{
   private String message;

   protected Hello(String message) throws RemoteException {
       this.message = message;
   }

   public String say(Client c) throws RemoteException {
       return this.message + c.getNom();
   }
```

-

# Passage des arguments : Exercice 2

Développer un couple de processus ($P_A$, $P_B$), tel que :

- $P_A$ crée un objet A.
- $P_B$ crée un objet B.
- Le processus $P_A$ appel une méthode $B.m(..)$.
- La méthode $B.m(..)$ rappel une méthode de A.

Proposer une solution distribuée, tel que :

- Chaque processus crée un serveur de noms et enregistre son objet (corriger dans le cours précédent).
- L'objet $A$ est passé en paramètre (référence/valeur) de la méthode $m()$ de $B$.

# Les Exceptions en RMI .

# Les Exceptions en RMI

- Les exceptions sont très **importantes** et cela pour une bonne qualité de service.

# Les Exceptions en RMI

- Les exceptions sont très **importantes** et cela pour une bonne qualité de service.
- Les exceptions sont **sérializable** en RMI, elle sont envoyées comme retour d'appel de méthodes.

# Les Exceptions en RMI

- Les exceptions sont très **importantes** et cela pour une bonne qualité de service.
- Les exceptions sont **sérializable** en RMI, elle sont envoyées comme retour d'appel de méthodes.
- Si un client appelle un serveur inexistant, il obtiendra une exception.

# Les Exceptions en RMI

RMI définit des exceptions spécifiques :

- **NotBoundException** : La référence appelée dans l'annuaire n'existe pas.

# Les Exceptions en RMI

RMI définit des exceptions spécifiques :

- **NotBoundException** : La référence appelée dans l'annuaire n'existe pas.
- **AlreadyBoundException** : La référence est déjà liée (existe déjà).

# Les Exceptions en RMI

RMI définit des exceptions spécifiques :

- **NotBoundException** : La référence appelée dans l'annuaire n'existe pas.
- **AlreadyBoundException** : La référence est déjà liée (existe déjà).
- **RemoteException** : exception liée au fonctionnement du bus logiciel.

# Les Exceptions en RMI

RMI définit des exceptions spécifiques :

- **NotBoundException** : La référence appelée dans l'annuaire n'existe pas.
- **AlreadyBoundException** : La référence est déjà liée (existe déjà).
- **RemoteException** : exception liée au fonctionnement du bus logiciel.
- **MalformedURLException** : URL donnée pour obtenir l'objet distant est mal formée.

# Les Exceptions en RMI

```
public static void main(String[] args) {
    Model modele;
    try {
        modele = new Model();
        LocateRegistry.createRegistry(1099);
        Naming.bind("rmi://localhost:1099/topModel", modele);
        System.out.println("serveur modele connecté");
        EntrepotInterface entrepot = (EntrepotInterface)
                Naming.lookup("rmi://localhost:1098/entrepot");
        entrepot.stockage(modele);
    }catch (RemoteException e) {
        System.out.println("Erreur de connexion au serveur entrepôt");
        e.printStackTrace();
    }
    catch (MalformedURLException e) {
        System.out.println("Erreur sur le nom logique de l'objet distant");
        e.printStackTrace();
    } catch (AlreadyBoundException e) {
        System.out.println("La référence de l'objet distant modele existe déja");
        e.printStackTrace();
    }catch (NotBoundException e) {
        System.out.println("La référence de l'objet distant entrepot n'existe pas");
        e.printStackTrace();
    }
```

# RMI en multithreads. .

# RMI en multithreads

- Les objets serveurs en RMI peuvent répondre à **plusieurs** client **simultanément**.

# RMI en multithreads

- Les objets serveurs en RMI peuvent répondre à **plusieurs** client **simultanément**.
- Les méthodes offertes à distance sont en mode **multithreads**.

# RMI en multithreads

- Les objets serveurs en RMI peuvent répondre à **plusieurs** client **simultanément**.
- Les méthodes offertes à distance sont en mode **multithreads**.
- Il faut géré les accès concurrents aux attributs et méthodes.

# RMI en multithreads

- Les objets serveurs en RMI peuvent répondre à **plusieurs** client **simultanément**.
- Les méthodes offertes à distance sont en mode **multithreads**.
- Il faut géré les accès concurrents aux attributs et méthodes.
- Il faut identifié les attributs utilisés dans les méthodes distantes.

# RMI en multithreads

- Les objets serveurs en RMI peuvent répondre à **plusieurs** client **simultanément**.
- Les méthodes offertes à distance sont en mode **multithreads**.
- Il faut géré les accès concurrents aux attributs et méthodes.
- Il faut identifié les attributs utilisés dans les méthodes distantes.
- La gestion des threads sur le serveur :

# RMI en multithreads

- Les objets serveurs en RMI peuvent répondre à **plusieurs** client **simultanément**.
- Les méthodes offertes à distance sont en mode **multithreads**.
- Il faut géré les accès concurrents aux attributs et méthodes.
- Il faut identifié les attributs utilisés dans les méthodes distantes.
- La gestion des threads sur le serveur :
  - En coordonnant les threads.

# RMI en multithreads

- Les objets serveurs en RMI peuvent répondre à **plusieurs** client **simultanément**.
- Les méthodes offertes à distance sont en mode **multithreads**.
- Il faut géré les accès concurrents aux attributs et méthodes.
- Il faut identifié les attributs utilisés dans les méthodes distantes.
- La gestion des threads sur le serveur :
  - En coordonnant les threads.
  - En mettant des priorités sur les threads.

# RMI en multithreads

Utiliser l'exclusion mutuelle pour éviter des accès concurrent à un attribut.

- Coté serveur :

# RMI en multithreads

Utiliser l'exclusion mutuelle pour éviter des accès concurrent à un attribut.

- Coté serveur :
  - Verrouiller l'accès concurrent à un attribut ou une méthode en utilisant **synchronized**.

# RMI en multithreads

Utiliser l'exclusion mutuelle pour éviter des accès concurrent à un attribut.

- Coté serveur :
  - Verrouiller l'accès concurrent à un attribut ou une méthode en utilisant **synchronized**.
  - Ce verrou sera posé sur l'objet distant en commun à tout les clients.

# RMI en multithreads

Utiliser l'exclusion mutuelle pour éviter des accès concurrent à un attribut.

- Coté client :

# RMI en multithreads

Utiliser l'exclusion mutuelle pour éviter des accès concurrent à un attribut.

- Coté client :
  - Le client peut aussi synchroniser l'objet distant en utilisant des méthodes **synchronized**.

# RMI en multithreads

Utiliser l'exclusion mutuelle pour éviter des accès concurrent à un attribut.

- Coté client :
  - Le client peut aussi synchroniser l'objet distant en utilisant des méthodes **synchronized**.
  - Ce verrou est posé sur le stub et non sur l'objet distant.

# RMI en multithreads

Utiliser l'exclusion mutuelle pour éviter des accès concurrent à un attribut.

- Coté client :
  - Le client peut aussi synchroniser l'objet distant en utilisant des méthodes **synchronized**.
  - Ce verrou est posé sur le stub et non sur l'objet distant.
  - Cette synchronisation est local au contexte du client, et non partagée par les autres clients.

# RMI en multithreads

Utiliser l'exclusion mutuelle pour éviter des accès concurrent à un attribut.

- Coté client :
  - Le client peut aussi synchroniser l'objet distant en utilisant des méthodes **synchronized**.
  - Ce verrou est posé sur le stub et non sur l'objet distant.
  - Cette synchronisation est local au contexte du client, et non partagée par les autres clients.
  - Cela permet de synchroniser les différents appels distants d'un même client.

# RMI en multithreads

Utiliser l'exclusion mutuelle pour éviter des accès concurrent à un attribut.

- Coté client :
  - Le client peut aussi synchroniser l'objet distant en utilisant des méthodes **synchronized**.
  - Ce verrou est posé sur le stub et non sur l'objet distant.
  - Cette synchronisation est local au contexte du client, et non partagée par les autres clients.
  - Cela permet de synchroniser les différents appels distants d'un même client.

```
modele = (ModelInterface)
        Naming.lookup("rmi://localhost:1099/topModel");
Date uneDate;
synchronized (modele) {uneDate = modele.datation();}
```

# Autres aspects de RMI

RMI propose plusieurs service afin d'améliorer la qualité de ses applications :

- Le service d'activation :

# Autres aspects de RMI

RMI propose plusieurs service afin d'améliorer la qualité de ses applications :

- Le service d'activation :
  - elle évite de maintenir des serveurs actifs en mémoire en attente de client,

# Autres aspects de RMI

RMI propose plusieurs service afin d'améliorer la qualité de ses applications :

- Le service d'activation :
  - elle évite de maintenir des serveurs actifs en mémoire en attente de client,
  - un démon java va créer l'objet distant seulement quand un client se connecte.

# Autres aspects de RMI

RMI propose plusieurs service afin d'améliorer la qualité de ses applications :

- Le service d'activation :
  - elle évite de maintenir des serveurs actifs en mémoire en attente de client,
  - un démon java va créer l'objet distant seulement quand un client se connecte.
  - Pour cela la classe **Activable** est utilisée.

# Autres aspects de RMI

RMI propose plusieurs service afin d'améliorer la qualité de ses applications :

- Le service d'activation :
  - elle évite de maintenir des serveurs actifs en mémoire en attente de client,
  - un démon java va créer l'objet distant seulement quand un client se connecte.
  - Pour cela la classe **Activable** est utilisée.
- RMI IIOP :

# Autres aspects de RMI

RMI propose plusieurs service afin d'améliorer la qualité de ses applications :

- Le service d'activation :
  - elle évite de maintenir des serveurs actifs en mémoire en attente de client,
  - un démon java va créer l'objet distant seulement quand un client se connecte.
  - Pour cela la classe **Activable** est utilisée.
- RMI IIOP :
  - Un objet RMI peut être mis à disposition de d'autres applications cliente CORBA.