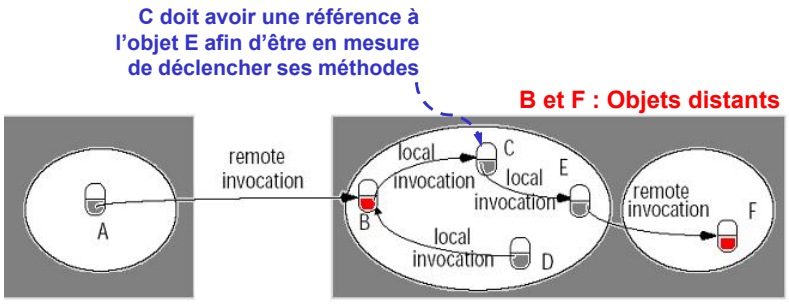
Communication Inter-processus

Présentation des caractéristiques des protocoles de communication interprocessus :

- Communication par objet: RMI, …
- Communication asynchrone RMI (asynchrone)
- Sérialisation d'objets
- Lectures et références

RMI (Remote Method Invocation)

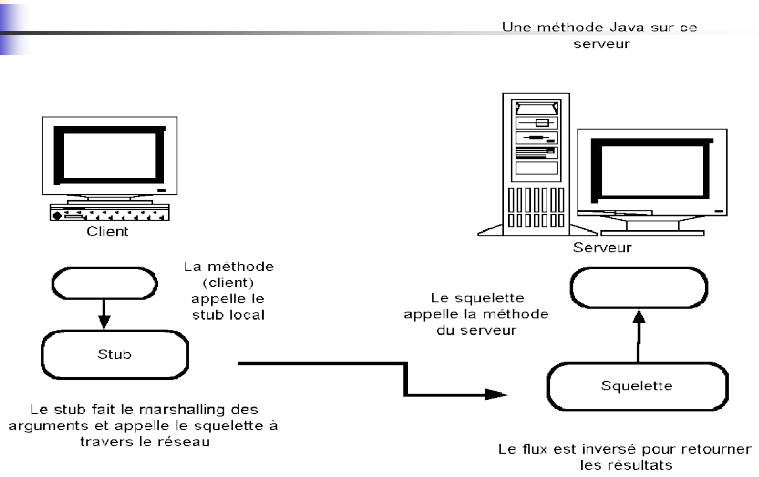
- **Modèle d'objets répartis :** permet aux objets de différents processus de se communiquer entre eux par appel de méthodes à distance



Introduction

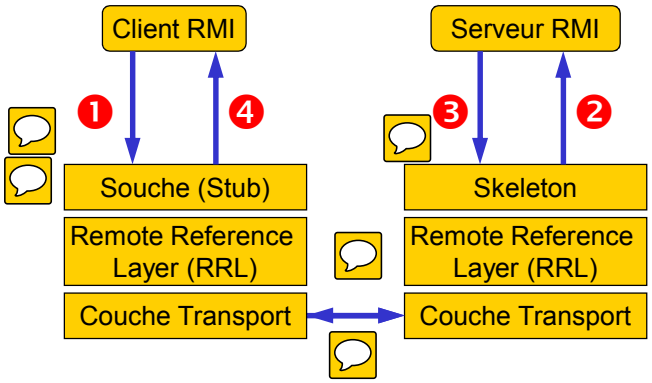
- RMI :
  - permet d'invoquer des méthodes d'objets distribués sur un réseau Internet.
  - est une approche semblable à l'approche RPC.
  - est un ensemble d'outils et de classes qui rendent l'implantation d'appels de méthodes et d'objets distants aussi simple que leur implémenttation dans un contexte local.
  - est basées sur la notion de souche et squelette (stub/skeleton).
- Les connexions et les transferts de données dans RMI sont effectués par Java sur TCP/IP grâce à un protocole propriétaire (JRMP, Java Remote Method Protocol) sur le port 1099.
- A partir de Java 2 version 1.3, les communications entre client et serveur s'effectuent grâce au protocole RMI-IIOP (Internet Inter-Orb Protocol), un protocole normalisé par l'OMG (Object Management Group) et utilisé dans l'architecture CORBA.

Comment le faire avec RMI?



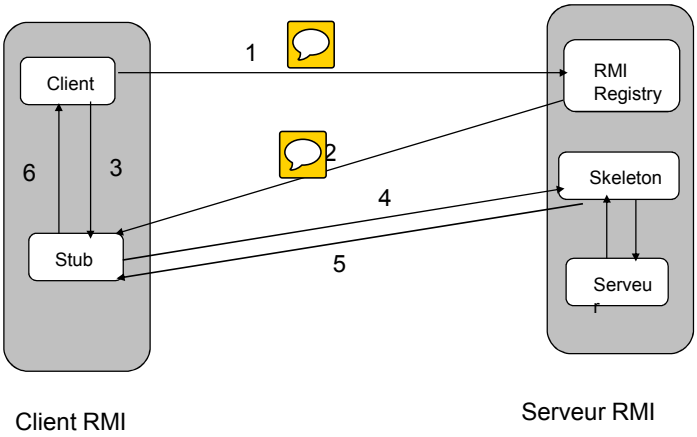
RMI (Remote Method Invocation)

- **Architecture :**



Modèle RMI

- L'architecture de RMI est schématisée ci-dessous :



Modèle RMI

- Lorsqu'un objet instancié sur une machine cliente désire accéder à des méthodes d'un objet distant, il effectue les opérations suivantes :
  1. il localise l'objet distant grâce à un service de désignation : le registre RMI
  2. il obtient dynamiquement une image virtuelle de l'objet distant (appelée stub ou souche en français). Le stub possède exactement la même interface que l'objet distant.
  3. Le stub transforme l'appel de la méthode distante en une suite d'octets, c'est ce que l'on appelle la sérialisation, puis les transmet au serveur instanciant l'objet sous forme de flot de données. On dit que le stub "marshalise" les arguments de la méthode distante.
  4. Le squelette instancié sur le serveur "désérialise" les données envoyées par le stub (on dit qu'il les "démarshalise"), puis appelle la méthode en local
  5. Le squelette récupère les données renvoyées par la méthode (type de base, objet ou exception) puis les marshalise
  6. le stub démarshalise les données provenant du squelette et les transmet à l'objet faisant l'appel de méthode à distance

Développement d'applications RMI

- Pour créer une application avec RMI il suffit de procéder comme suit :
  1. définir l'interface pour la classe distante. Celle-ci doit implémenter l'interface *java.rmi.Remote* et déclarer les méthodes publiques globales de l'objet, c'est-à-dire les méthodes partageables. De plus ces méthodes doivent pouvoir lancer une exception de type *java.rmi.RemoteException*.
  2. définir la classe distante. Celle-ci doit dériver de *java.rmi.server.UnicastRemoteObject* (utilisant elle-même les classes *Socket* et *SocketServer*, permettant la communication par protocole TCP). De plus cette classe doit implanter l'interface définie dans l'étape 1.
  3. Créer un programme client capable d'accèder aux méthodes d'un objet sur le serveur grâce à la méthode *Naming.lookup()*
  4. Créer les classes pour le stub et le squelette grâce à la commande *rmic*
  5. Lancer le registre RMI et lancer l'application serveur, c'est-à-dire instancier l'objet distant. Celui-ci lors de l'instanciation créera un lien avec le registre
  6. Compiler l'application cliente
  7. Instancier le client

Définition de l'interface

- L'interface utilisée doit hériter de l'interface Remote. Ses méthodes doivent déclenchées des exception de type RemoteException.
- Par exemple, l'interface PiRemote hérite de l'interface Remote et déclare la méthode getPi(), qui déclenché une execption :

```
import java.rmi.*;

interface PiRemote extends Remote {
                    double getPi() throws RemoteException;

}
```

Implémentation et exportation des objets (classe distante)

- Cette classe distante implémente l'interface PiRemote et hérite de la classe UnicastRemoteObject :

```
import java.net.*;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class Pi extends UnicastRemoteObject  implements PiRemote {

        public double getPi() throws RemoteException {
                return Math.PI;
        }
        public Pi( )  throws RemoteException { }
     public static void main(String args) {

                System.setSecurityManager(new RMISecurityManager() );
                try {
          Pi p= new Pi(); // instancier la classe distante
          Naming.rebind("localhost:6543/MonPi",p);//lier cette instance à MonPi
                }
                catch (Exception e) {
         System.out.println("Erreur: "+e);
                }
     }
} //fin de la classe Pi
```

Construction du client et appel des méthodes

- Le client récupère une référence sur l'implémentation distante *via* le nom qui lui a été attribué précédemment, puis appelle sur cette référence la méthode distante, comme si elle était locale.

```
import java.rmi.*;
import java.rmi.registry.*;

public class OutputPi {
        public static void main(String[] args) {
             System.setSecurityManager(new RMISecurityManager());

             try {
                 PiRemote pr=
   (PiRemote)Naming.lookup("rmi://localhost:6543/MonPi");
                 for (int i =0; i < 10; i++)
                      System.out.println("Pi= " + pr.getPi() );
              } catch (Exception e) { System.out.println(e); }
        }
}
```

Compilation et exécution de l'application

- Compilation du serveur:
  rmic Pi.java : **Ce qui donne**: Pi.class, Pi_Skel.class et Pi_Stub.class
- Activation du registre:
  remiregistry 6543 &
- Exécution du serveur:
  java Pi &
- Compilation du client:
  javac OutputPi.java
- Exécution du client:
  java OutputPi

Exporter un objet

- Pour être exporté, un objet doit implémenter la classe Remote. Il y a alors deux possibilités pour être exporté:
  - hériter de la classe UnicastRemoteObject . L'appel au constructeur exportera automatiquement l'objet courant.
  - appeler la méthode exportObjet() de la classe UnicastRemoteObject sur l'objet à exporter. Cette méthode retourne l'objet RemoteStub associé.
- Lorsqu'un objet est exporté, un ensemble de *thread*s est créé pour attendre les appels des méthodes.

Le service de nommage

- Le service de nommage permet de récupérer le *Stub* d'un objet distant à partir d'un nom.
- La classe Naming permet d'accéder de façon simple au service de nommage d'une machine en utilisant un format de type URL:

  rmi://machine:port/objet
- Avant de pouvoir enregistrer un objet il faut lancer un serveur de nommage au moyen de:

  rmiregistry port &
- Il est alors possible d'enregistrer un objet local au moyen de bind(String name, Remote obj) et rebind(String name, Remote obj)

Le service de nommage

- Il est possible de récupérer le *Stub* d'un objet distant à partir de son URL au moyen de la méthode Remote lookup(String name).
- Les méthodes suivantes sont utilisées:
  - unbind(String name) pour désenregistrer l'objet du service de nommage,
  - list(String name) pour lister l'ensemble des objets enregistrés.

Passage de paramètres et retours

- Les variables de type primitif sont passées par *valeur*.
- Les objets qui n'implémentent pas l'interface Remote sont passés par *copie/restauration* à condition qu'ils implémentent l'interface Serializable ou Externalizable.
- Les objets qui implémentent l'interface Remote (*Référence*) sont remplacés par l'objet *Stub* correspondant lors du passage de paramètres.
- Le retour de valeur a la même sémantique.

RMI (Remote Method Invocation) (4)

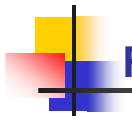
- **Programmation avec RMI :** étapes à suivre
  1) **Définir les interfaces pour les classes distantes**
  2) **Créer et compiler les implémentations de ces classes**
  3) **Créer les classes pour la souche et le skeleton : RMIC**
  4) **Créer et compiler une application serveur**
  5) **Lancer RMIREGISTRY et lancer l'application serveur**
  6) **Créer et compiler un programme client qui accède aux objets distants du serveur**
  7) **Lancer ce client**

RMI (Remote Method Invocation) (5)

- **Étude de cas :**
  - Un gestionnaire de comptes bancaires capable de gérer plusieurs comptes
  - Il sera placé sur une machine distante, interrogé et manipulé par des clients

1) Définir les interfaces pour les classes distantes

- **Classe CreditManager :** rechercher ou créer un compte bancaire
- **Classe CreditCard :** faire des achats, poser une signature, ou avoir l'état du compte



RMI (Remote Method Invocation) (6)

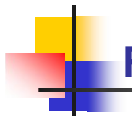
- **Classe CreditManager :**

```
package credit;
import credit.*;
import java.rmi.*;

public interface CreditManager extends Remote {

   public CreditCard findCreditAccount(String Customer)
          throws UnknownAccountException, RemoteException;

   public CreditCard newCreditAccount(String newCustomer)
          throws DuplicateAccountException, RemoteException;
}
```

RMI (Remote Method Invocation) (7)

- **Classe CreditCard :**

```
package credit;
import credit.*;
import java.rmi.*;

public interface CreditCard extends Remote {
   public float getCreditLine() throws RemoteException;
   public void makePurchase(float amount, int signature)
          throws  InvalidSignatureException, RemoteException,
                  CreditLineExceededException;
   public void setSignature(int pin) throws RemoteException;
}
```

RMI (Remote Method Invocation) (8)

2) Créer et compiler les implémentations de ces classes

```
package credit;
import java.rmi.*;
import java.rmi.server.*;
import java.util.Hashtable;

public class CreditManagerImpl extends UnicastRemoteObject
                                implements CreditManager {
   private static transient Hashtable accounts = new Hashtable();
   public CreditManagerImpl() throws RemoteException { }
…
```


RMI (Remote Method Invocation) (9)

2) Créer et compiler les implémentations de ces classes (suite)

```
public CreditCard newCreditAccount(String customerName)
        throws DuplicateAccountException, RemoteException {

    CreditCardImpl newCard = null;
    if (accounts.get(customerName) == null)
        newCard = new CreditCardImpl(customerName);
    else throw new DuplicateAccountException();
    accounts.put(customerName, newCard);
    return newCard;
}
…
```

RMI (Remote Method Invocation) (9)

2) Créer et compiler les implémentations de ces classes (suite)

```
public CreditCard findCreditAccount(String customer)
        throws UnknownAccountException, RemoteException {
    CreditCardImpl
        account = (CreditCardImpl)accounts.get(customer);
     if (account == null)
        throw new UnknownAccountException();
     else
        return account;
}
}
```


RMI (Remote Method Invocation) (10)

2) Créer et compiler les implémentations de ces classes (suite)

```
package credit;
import java.rmi.*;
import java.rmi.server.*;
import java.io.Serializable;

public class CreditCardImpl extends UnicastRemoteObject
                            implements CreditCard, Serializable
{
        private float creditLine = 5000f;
        private int signature = 0;
        private String accountName;
…
```

RMI (Remote Method Invocation) (11)

2) Créer et compiler les implémentations de ces classes (suite)

```
public CreditCardImpl(String customer)
                                        throws RemoteException {
    accountName = customer;
}
public float getCreditLine() throws RemoteException {
    return creditLine;
}
public void setSignature(int pin) throws RemoteException {
     signature = pin;
}
…
```


RMI (Remote Method Invocation) (12)

2) Créer et compiler les implémentations de ces classes (suite)

```
public void makePurchase(float amount, int signature)
          throws  InvalidSignatureException, RemoteException,
                  CreditLineExceededException {
    if (signature != this.signature)
          throw new InvalidSignatureException();
    if (amount > creditLine)
          throw new CreditLineExceededException();
    else
          creditLine -= amount;
}
}
```

RMI (Remote Method Invocation) (13)

4) Créer et compiler une application Serveur

- **RMIregistry (**classeur de Java RMI) **:** maintien une table des références d'objets distants
- **classe Naming**
  - void **rebind** (String name, Remote obj)
  - void **bind** (String name, Remote obj)
  - void **unbind** (String name, Remote obj)
  - Remote **lookup**(String **name**)
  - String [] **list**()

//computerName:port/objectName

Indique où se trouve RMIregistry

RMI (Remote Method Invocation) (13.1)

4) Créer et compiler une application Serveur

```
package credit;
import java.util.*;
import java.rmi.*;

public class ServerBank {
    public static void main (String args[]) {
        System.setSecurityManager(new RMISecurityManager());
        try {
            CreditManagerImpl cmi = new CreditManagerImpl();
            Naming.rebind("cardManager", cmi);
        } catch (Exception e) {
            System.out.println(e.getMessage());
}}}
```

RMI (Remote Method Invocation) (14)

6) Créer et compiler une application Client

```
package credit;
import java.rmi.*;

public class ClientBank {
    public static void main(String args[]) {
        CreditManager cm = null;
        CreditCard account = null;
        if (args.length < 3) {
            System.err.println("Usage : java ClientBancaire" +
                               " <server> <name1> <name2>");
            System.exit (1);
        }
…
```


RMI (Remote Method Invocation) (15)

6) Créer et compiler une application Client (suite)

```
System.setSecurityManager(new RMISecurityManager());
try {
    String url = new String ("rmi://" + args[0] + "/cardManager");
    cm = (CreditManager)Naming.lookup(url);
} catch (Exception e) {
    System.out.println("Erreur a l'accès du gest. banc." + e);
}
try {
     account = cm.newCreditAccount(args[1]);
} catch (Exception e) {
    System.out.println("Erreur de création " + args[1] + " : " + e);
} …
```

RMI (Remote Method Invocation) (16)

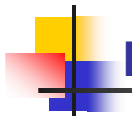
6) Créer et compiler une application Client (suite)

```
try {
    System.out.println("crédit disp. : " + account.getCreditLine());
    account.setSignature(1234);
    account.makePurchase(100.00f, 1234);
    System.out.println("crédit disp. : " + account.getCreditLine());
    account.makePurchase(160.00f, 1234);
    System.out.println("crédit disp. : "+ account.getCreditLine());
} catch (Exception e) {
     System.out.println("Erreur de transactions du " + args[1]);}
try {
     account = cm.findCreditAccount(args[2]);
} catch (Exception e) {
     System.out.println("Erreur de recherche  " + args[2] + e);}}
```


RMI (Remote Method Invocation) (17)

7) Compiler et exécuter l'application

- javac -d . -classpath . *.java
- rmic -d . -classpath . credit.CreditCardImpl credit.CreditManagerImpl
- start rmiregistry
- rmic -d . -classpath . credit.CreditCardImpl credit.CreditManagerImpl
  - CreditCardImpl_Skel.class
  - CreditCardImpl_Stub.class
  - CreditManagerImpl_Skel.class
  - CreditManagerImpl_Stub.class
- start java -Djava.security.policy=c:\BanqueRMI\permit.policy -Djava.rmi.server.codebase=file:///c:/BanqueRMI/ -classpath . credit.ServerBank
- java -Djava.security.policy=c:\BanqueRMI\permit.policy -Djava.rmi.server.codebase=file:///c:/BanqueRMI/ -classpath . credit.ClientBank 209.197.157.169 Pierre Marie

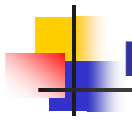


RMI (Remote Method Invocation) (17)

7) Compiler et exécuter l'application

Résultat d'exécution :

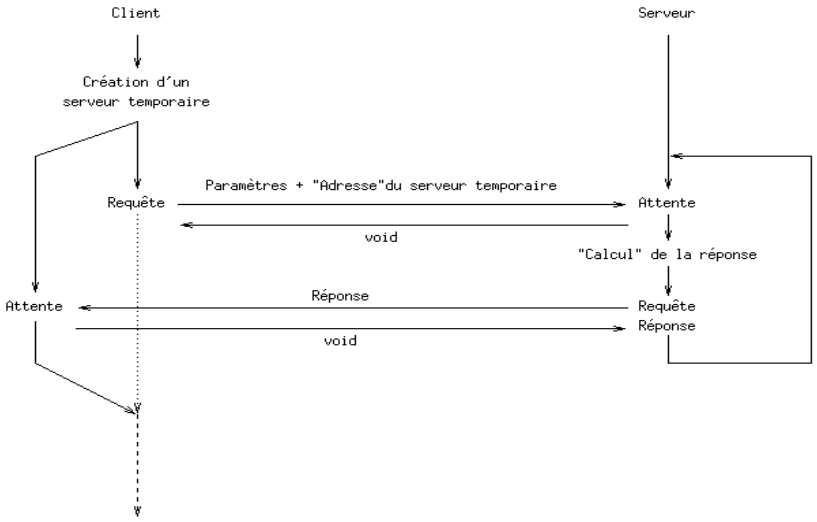
```
crédit disp. : 5000.0
Changement de signature
un achat de 100 $
crédit disp. : 4900.0
second achat de 160 $
crédit disp. : 4740.0
Erreur de recherche  Marie :
                    credit.UnknownAccountException
```

RMI (Remote Method Invocation) (18)

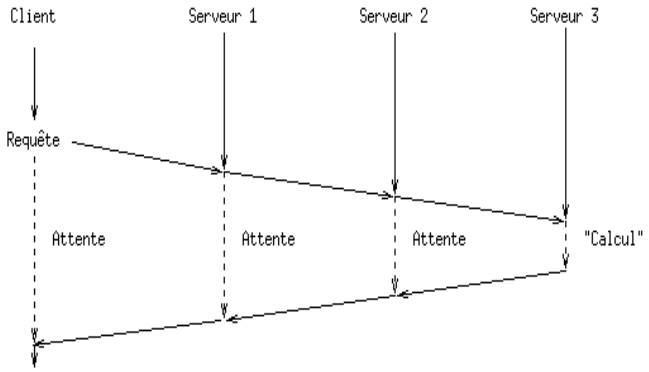
- **Invocation asynchrone :**
  - **RMI :** invocations de méthodes synchrones (tant que la méthode distance n'est pas terminée, l'appelant est bloqué)
  - **Solution :** fournir, du côté client, un objet distant (OD) temporaire pour la réception des résultats (schéma)
  - **Avantage :** les invocations asynchrones sont particulièrement profitables dans le cas d'appel en cascade (schéma)

Invocations asynchrones
Principes



Invocations asynchrones
Principes

- Les invocations asynchrones sont particulièrement profitables dans le cas d'appel en cascade



Invocations asynchrones : exemple 1

- Pour réaliser un appel RMI asynchrone au niveau client, il suffit d'encapsuler l'appel dans un thread comme illustré l'exemple suivant :
- Ecrire un serveur RMI qui propose une méthode echo classique (elle renvoie au client la chaîne de caractères paramètre de l'appel), mais avec une pause importante au début de la méthode, pour simuler un calcul complexe (on pourra tirer au hasard la durée de la pause).
- Ecrire un client pour ce serveur qui réalise l'appel de façon asynchrone en procédant comme suit :
  - écrire un objet Thread (ou Runnable) qui réalise l'appel puis affiche le résultat ;
  - démarrer l'appel dans le thread principal et exécuter en parallèle (toujours dans ce thread principal) des opérations quelconques (comme par exemple quelques affichages entrecoupés de pauses).

Interface : IServeur.java

```
import java.rmi.*;
public interface IServeur extends Remote {
  public String echo(String msg) throws RemoteException;
}
```

Le serveur : Serveur.java

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
public class Serveur extends UnicastRemoteObject implements IServeur {
  public Serveur() throws RemoteException {
    super();
  }
  public String echo(String msg) throws RemoteException {
    System.out.println("Ping");
    try {
      Thread.sleep(5000);
    } catch(InterruptedException e) {
    }
    System.out.println("Pong");
    return "["+msg+"]";
  }
  public static void main(String[] args) throws Exception {
    // démarrage du serveur
    Serveur serveur = new Serveur();
    Naming.rebind("ServeurEcho",serveur);
    System.out.println("Serveur démarré");
  }
}
```

Le client : Client.java

```
import java.rmi.*;
public class Client {
  public static void main(String[] args) throws Exception {
    // le final est ajouté à cause de la classe anonyme
    final IServeur serveur = (IServeur)Naming.lookup("ServeurEcho");
    // solution avec une classe anonyme
    Thread t = new Thread(new Runnable() {
    public void run() {
      try {
        System.out.println(serveur.echo("coucou"));
      } catch(RemoteException e) {
        System.out.println(e);
      }
    }
      });
    t.start();
    System.out.println("démarrage de l'appel");
    for(int i=1;i<=5;i++) {
      System.out.println(i);
      try {
          Thread.sleep(1500);
      } catch (InterruptedException e) {
      }
    }
    System.out.println("fin du main");
  }
}
```

Invocations asynchrones : exemple 2

- Le problème de l'exemple est que c'est le thread qui exécute l'appel qui décide de faire l'affichage du résultat. Dans certaines situations, cela n'est pas vraiment acceptable, d'où l'exemple suivant :
- Modifier l'objet d'appel asynchrone de l'exercice précédent afin d'obtenir les fonctionnalités suivantes :
  - la méthode run() ne doit plus afficher le résultat mais se contenter de le stocker dans une variable adaptée;
  - l'objet doit proposer une méthode permettant de savoir si le calcul est terminé;
  - l'objet doit proposer une méthode bloquante renvoyant le résultat de l'appel (le blocage sera basé sur le couple wait/notify)

Interface : IServeur.java

```
import java.rmi.*;
public interface IServeur extends Remote {
  public String echo(String msg) throws
    RemoteException;
}
```

Le serveur : Serveur.java

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
public class Serveur extends UnicastRemoteObject implements IServeur {
  public Serveur() throws RemoteException {
    super();
  }
  public String echo(String msg) throws RemoteException {
    System.out.println("Ping");
    try {
      Thread.sleep(5000);
    } catch(InterruptedException e) {
    }
    System.out.println("Pong");
    return "["+msg+"]";
  }
  public static void main(String[] args) throws Exception {
    // démarrage du serveur
    Serveur serveur = new Serveur();
    Naming.rebind("ServeurEcho",serveur);
    System.out.println("Serveur démarré");
  }
}
```

Le client : Client.java

```
import java.rmi.*;
public class Client {
  public static void main(String[] args) throws Exception {
    IServeur serveur = (IServeur)Naming.lookup("ServeurEcho");
    AppelAsynchrone appel = new AppelAsynchrone(serveur,"coucou");
    appel.start();
    System.out.println("démarrage de l'appel");
    for(int i=1;i<=5;i++) {
      System.out.println(i);
      try {
      Thread.sleep(500);
      } catch (InterruptedException e) {
      }
    }
    System.out.println("Attente du résultat");
    System.out.println(appel.getResult());
    System.out.println("fin du main");
  }
}
```

Le client : AppelAsynchrone.java

```
import java.rmi.*;
public class AppelAsynchrone extends Thread {
  private IServeur serveur;
  private boolean done;
  private String result;
  private String param;
  public                AppelAsynchrone(IServeur
       serveur,String param) {
    this.serveur = serveur;
    done = false;
    this.param = param;
  }
  public void run() {
    try {
       result = serveur.echo(param);
    } catch(RemoteException e) {
      System.out.println(e);
    }
    synchronized(this) {
      done = true;
      notify();
    }
  }
```

```
public synchronized boolean isDone() {
      return done;
}
  public synchronized String getResult() {
    while(!done) {
      try {
       wait();
      } catch (InterruptedException e) {
      }
    }
    return result;
  }
}
```

RMI (Remote Method Invocation)

- **Invocation asynchrone (suite) : principes**
  - OD temporaire : construit de la même façon que les autres ODs
  - Exporté de la même façon
  - Il n'est pas nécessaire de le référencer dans `Naming`
  - Il doit implémenter `Unreferenced` pour se terminer correctement

RMI (Remote Method Invocation)

- **Invocation asynchrone (suite) : exemple**

OD du Serveur Temporaire

```
public interface ODTmpServer extends Remote {
  public void returnString(String text) throws RemoteException;
}
public class ODTmpServerImpl  implements ODTmpServer,
                                                  Unreferenced {
   public void returnString(String text) throws RemoteException {
       System.out.println(text);
   }
   public void unreferenced() {//Called when there are no current references to this remote object
       ThreadGroup g = Thread.currentThread().getThreadGroup();
       g.stop();
}}
```


RMI (Remote Method Invocation)

- **Invocation asynchrone : exemple (suite)**

OD du Serveur Principale

```
public interface ODServer extends Remote {
  public void call(String st, ODTmpServer tmpServer)
                                                  throws RemoteException;
}
public class ODServerImpl extends UnicastRemoteObject
                                                  implements ODServer {
  public ODServerImpl() throws RemoteException { }
  public void call(String text, ODTmpServer odTmpServer)
                                                  throws RemoteException {
    try {
          for(int i=1; i<1000; i++) System.out.println(i);
          odTmpServer.returnString(text.toUpperCase());
    } catch …}
```

RMI (Remote Method Invocation)

- **Invocation asynchrone : exemple (suite)**

Classe du serveur temporaire

```
public class TmpServer extends Thread {
  private ODTmpServerImpl odTmpServer;
  private ODServer DistantServer;
  public TmpServer (ODServer server) {
          DistantServer = server;
  }
  public void run() {
     odTmpServer = new ODTmpServerImpl();
     try {
          UnicastRemoteObject.exportObject(odTmpServer);
          DistantServer.call("Hello", odTmpServer);
     } catch (Exception e) {System.out.println("Erreur " + e);}
}}
```


RMI (Remote Method Invocation)

- **Invocation asynchrone : exemple (suite)**

Programme Serveur

```
package asynchrone;
import java.rmi.*;
public class Server  {
  public static void main(String[] args) {
     System.setSecurityManager(new RMISecurityManager());
     try {
          ODServerImpl server = new ODServerImpl();
          Naming.rebind("//localhost/Server", server);
          System.out.println("Serveur Prêt");
     } catch (Exception e) {
           System.out.println(e.getMessage());}
  }}
```

RMI (Remote Method Invocation)

- **Invocation asynchrone : exemple (suite)**

Programme Client

```
public class Client {
  public static void main(String[] args) {
    System.setSecurityManager(new RMISecurityManager());
    ODServer server = null;
    try {
      server =(ODServer)Naming.lookup("rmi://localhost/Server");
     } catch (Exception e) {System.out.println("Erreur " + e);}
    TmpServer tmpServer = new TmpServer( server);
    tmpServer.start();
    for(int i=1; i<10000; i++) System.out.println(i);
    tmpServer = null;
}}
```


Sérialisation

- Java permet l'accès aux données externes via des streams:
  - pour le stockage externe de données:
    - streams de caractères
    - streams d'octets
  - pour permettre aux données de faire partie de l'objet
    - stream d'objets
- Pour sauvegarder un objet (et ses données) sur un support externe (ex. disque) ou le transmettre (ex. par Internet) on doit le sérialiser.

L'interface Sérializable

- Pour qu'un objet soit sérialisable, il doit implanter l'interface Serializable:

  class Message implements Serializable { ... }

- La serialisation permet la *persistence* des objets:
  - La capacité d'un objet d'exister et de fonctionner en dehors du programme qui l'a créé.
- Quand un objet est sérialisé toutes les variable d'instance et objets qu'il contient sont également sérialisés.
- On peut demander que des variables d'instances ne soient pas sérialisées:

  public transient int cle = 55

Les streams d'objets

- Pour sauvegarder un objet, on utilise *ObjectOutputStream*:

```
FileOutputStream fo = new FileOutputStream("Fichier.obj");
ObjectOutputStream oo = new ObjectOutputStream (fo)
```

- On utilise le méthodes suivantes pour l 'écriture:
  - Pour les types de base:

    write(int), write(byte), write(byt[]), writeBoolean(boolean), writeByte(int), writeBytes(String), writeInt(int),...
  - Pour les objets complexes:

    writeObject(Object)

Exemple: Sérialisation

```
import java.io.*;
import java.util.*;
public class ObjetADisque {
 public static void main(String[] args) {
Message mess = new Message();
        String source  = "Jean Alain";
        String dest = "Abdel Obaid ";
        String[] lettre = {
                "Bonjour Alain",
                "Je te rappelle que nous avons",
                "une réunion ce lundi",
                "On se rencontre au SH-4321",
                "Si tu as besoin de documents",
                "N hésite pas a me contacter",
                "Bien à toi",
                "Abdel" };
        Date quand = new Date();
mess.writeMessage( source , dest, quand,
lettre);
try {    FileOutputStream fo = new
                                FileOutputStream("Fichier.obj");
          ObjectOutputStream oo = new
                                ObjectOutputStream(fo);
                oo.writeObject(mess);
                oo.close();
                System.out.println("C'est fait !");
  } catch (IOException e) {;}
 }
}

class Message implements Serializable {
        int lignes;
        String De;
        String A;
        Date quand;
        String[] lettre;
        void writeMessage(
                String De_par, String A_par,
                Date quand_par,  String[] lettre_par)  {
                lettre = new String[lettre_par.length];
                for (int i =0 ; i < lettre_par.length; i++)
                        lettre[i]= lettre_par[i];
                lignes=lettre_par.length;
                A=A_par;  De=De_par;  quand=quand_par;
        }
}
```

Désérialisation

- Pour récupérer un objet qui a été sérialisé, on utilise des streams de lecture ObjectInputStream:

```
FileInputStream fi = new FileInputStream("Fichier.obj");
ObjectInputStream oi = new ObjectInputStream(fi);
```

- On utilise les méthodes suivantes pour la lecture:
  - Pour les types de base:

    read(), read(byte), readBoolean(),
    readChar(), readDouble(), readInt(),...
  - Pour les objets complexes qu'on doit caster:

    readObject(Object)

Références et lectures

- Coulouris et al., Distributed Systems, edition 4, 2005 (Chapitres 4 et 5).
- Andrew S. Tanenbaum, Distributed systems, 2002, Prentice-Hall, new Jersey.
- Tutoriel : ***Getting Started Using RMI,*** http://java.sun.com/j2se/1.3/docs/guide/rmi/getstart.doc.html
- Jim Waldo, ***Remote procedure calls and Java Remote Method Invocation***, IEEE Concurrency, 6(3), pages 5-7, Sept 1998
- A.D. Birrell and B.J. Nelson, ***Implementing Remote Procedure Calls (RPC)***, ACM Trans. on Computer Systems, 1984, pp 39-54