Langage Java et systèmes distribués

UIC – 2ème GI

Karim GUENNOUN

Points abordés

- Héritage et Interfaces
- Threads
- Communication par sockets
- RMI

Héritage

Concept

- L'héritage représente la relation: EST-UN
- Exemples:
  - Un Technicien est un Employé…
- La classe dont on dérive est dite classe de base ou classe mère :
  - Employé est la classe de base (classe supérieure),
- les classes obtenues par dérivation sont dites classes dérivées ou classes filles :
  - Technicien, Ingenieur et Directeur sont des classes dérivées (sous-classes).

Exemple

- package Test2018;
- public class Employe
- {
- String Nom;
- String Prenom;
- double Salaire;
- public String Afficher()
- {
- String s="Nom: "+ nom+", Prenom: "+ prenom+ " salaire: " + salaire;
- return(s); } }

Intérêt 1/2

- La réutilisation:
  - Profiter d'une classe déjà codée pour en définir une nouvelle
  - Optimisation de code: Pas besoin de reprendre les attributs et méthodes définis dans la classe mère.
  - package Test2018;
  - public class ChefService extends Employe
  - {
  - String NomService;
  - double PrimeEncadrement;

Intérêt 2/2

- La factorisation de code:
  - Factoriser des attributs et des méthodes en communs entre certaines classes
  - Ecrire ces attributs/méthodes une seule fois
  - Exemple: ChefDivision / ChefService

Classe dérivée

- Une classe dérivée modélise un cas particulier de la classe de base, et est enrichie d'informations supplémentaires.
- La classe dérivée possède les propriétés suivantes:
  - contient les attributs de la classe de base,
  - peut posséder de nouveaux attributs,
  - possède les méthodes de sa classe de base
  - peut redéfinir (masquer) certaines méthodes,
  - peut posséder de nouvelles méthodes

Héritage en Java

- Syntaxe: public class B **extends** A
- Une classe ne peut hériter que d'une seule classe à la fois. Il n'existe pas d'héritage multiple.
- Toutes les classes héritent de la classe Object. (package lang)
  - clone, equals et toString, getClass
  - Tout objet en Java peut être utilisé comme un verrou : wait, notify,…
  - ...

Le constructeur

- La classe dérivée doit prendre en charge la construction de la classe de base.
- Pour construire un Directeur, il faut construire d'abord un Employé
- Le constructeur de la classe de base est donc appelé avant le constructeur de la classe dérivée.
- Si un constructeur de la classe dérivée appelle explicitement un constructeur de la classe de base, cet appel doit être obligatoirement la première instruction de constructeur. Il doit utiliser pour cela, le mot clé **super**.

Héritage des constructeurs

- B hérite de A:
  - La classe A ne possède aucun constructeur ou possède un constructeur sans paramètre. Ce constructeur est alors appelé implicitement par défaut dans tous les constructeurs de B.
  - La classe A ne possède pas de constructeur par défaut et il n'existe que des constructeurs avec des paramètres. Alors, les constructeurs de B doivent appeler explicitement un des constructeurs de A.
  - L'appel super correspond forcément à la première ligne de code.

Droit d’accès

- Rappel, l’unité de protection est la classe:
  - public: les membres sont accessibles à toutes les classes,
  - «rien»: les membres sont accessibles à toutes les classes du même paquetage,
  - private: les membres ne sont accessibles qu'aux membres de la classe.
- Si les membres de la classe de base sont :
  - public ou « rien » : les membres de la classe dérivée auront accès à ces membres (champs et méthodes),
  - private : les membres de la classe dérivée n'auront pas accès aux membres privés de la classe de base (utiliser protected dans ce cas)

Les méthodes

- Redéfinition
  - Substituer une méthode par une autre
  - Même nom, même signature, même type de retour
  - public class ChefService extends Employe
  - {
  - String NomService;
  - double primeEncadrement;
  - public String Afficher()
  - {
  - String s=super.Afficher()+", NomService: "+NomService;
  - return(s); } }

Les méthodes

- Surdéfinition
  - Cumuler plusieurs méthodes ayant le même nom
  - Même nom mais signature différente
  - Ne doit pas diminuer les droits d'accès
    - …
    - public void Afficher(String Date)
    - {
    - System.out.println ("Nom: "+ nom+", Prenom: "+ prenom+ " salaire: " + salaire + ()+", NomService: "+NomService);
    - }
    - }

Compatibilité entre objets classe mère et classe fille

- Un objet de la classe fille est forcément un objet de classe mère
  - ChefService CS=new ChefService("toto", "titi",10000, "comptabilité",5000);
  - Employe E=new Employe(("toto", "titi",10000);
  - E=CS;
    - Conversion implicite: le compilateur recopie les attributs communs en ignorant le reste
  - CS=E;
    - Erreur: impossibilité de compléter les champs manquant
    - Obligation de caster pour passer la compilation: d=(Directeur) e;

Polymorphisme

- Permet de manipuler les objets sans connaitre leur type:
  - Employe[ ] tab=new Employe[3];
  - tab[0]=new Employe(….);
  - tab[1]=new Employe(…);
  - tab[2]=new ChefService(…);
  - for(i=0;i<0;i++)
  - {
  - System.out.println(tab[i].Afficher());
  - }

Méthodes et classe « final »

- Une méthode "final" ne peuvent être redéfinie dans les classes filles
- Les classes qui sont déclarées "final" ne peuvent posséder de classes filles

Récapitulatif

- Une classe hérite d'une autre classe par le biais du mot clé :extends.
- Une classe ne peut hériter que d'une seule classe.
- Si aucun constructeur n'est défini dans une classe fille, la JVM en créera un et appellera automatiquement le constructeur de la classe mère.
- La classe fille hérite de toutes les propriétés et méthodes public et protected.
- Les méthodes et les propriétés private d'une classe mère ne sont pas accessibles dans la classe fille.
- On peut redéfinir une méthode héritée
- On peut utiliser le comportement d'une classe mère par le biais du mot clé super.
- Si une méthode d'une classe mère n'est pas redéfinie ou « polymorphée », à l'appel de cette méthode par le biais d'un objet enfant, c'est la méthode de la classe mère qui sera utilisée.
- Vous ne pouvez pas hériter d'une classe déclarée final.
- Une méthode déclarée final n'est pas redéfinissable.

Classes abstraites et interfaces

Une classe abstraite, c'est quoi?

- Correspondent à des superclasses
- Utiliser principalement pour la conception ascendante
  - E.g : personne vis-à-vis : etudiant, enseignant, adminstratif
- Doit être déclarée avec le mot clé abstract
  - abstract class personne
- Permet de définir des méthodes sans donner leur corps
  - abstract void afficher();

Fonctionnement

- Une classe abstraite n’est pas instanciable
- Une méthodes abstraite ne peut être déclarée que dans une classe abstraite
- Les classes non abstraites héritant d’une classe abstraite doivent définir le comportement des méthodes abstraites
- S’il y en a plusieurs, le comportement est polymorphe

Une interface, c’est …

- Une interface est une classe 100% abstraite
- Une interface définit un contrat que certaines classes pourront s'engager à respecter.
- Une interface n’implémente pas les comportements, les comportements sont définis dans les classes d’implémentation
- Contient
  - des définitions de constantes
  - des déclarations de méthodes (prototype).

Interfaces et héritage

- L'héritage multiple est autorisé pour les interfaces
- Exemple
  - **interface** A { **void** f(); }
  - **interface** B **extends** A { **void** f1(); }
  - **interface** C **extends** A { **void** f2(); }
  - **interface** D **extends** B, C { **void** f3(); }

Implantation d’une interface

- Si une classe déclare qu'elle implémente une interface, elle doit proposer une implémentation des méthodes listées dans l'interface.
- La classe peut proposer des méthodes qui ne sont pas listées dans l'interface.

Implantation et héritage multiple

- Lorsqu'une interface hérite de plusieurs autres interfaces, il peut apparaître que des déclarations de méthodes de même nom sont héritées.
  - Si les deux déclarations de méthodes héritées ont des en-têtes identiques, il n'y a pas de problème.
  - Si les deux déclarations ont un même nom de méthode mais des paramètres différents, en types ou en nombre, la classe implémentant l'interface devra implémenter les deux méthodes.
  - Le problème apparaît lorsque deux déclarations de méthodes ont même nom et mêmes paramètres mais un type de retour différent. Dans ce cas, le compilateur Java refuse de compiler le programme car il y a un conflit de nom.

Quelques éléments…

- On ne peut pas instancier une interface en Java
- Le mot clé "**implements**" est utilisé pour indiquer qu'une classe implemente une interface
- L'implémentation de toute méthode appartenant à une interface doit être déclarée public
- La classe implementant une interface doit implémenter toutes les méthodes. Sinon, elle doit être déclarée abstract
- Les Interfaces ne peuvent être déclarées private ou protected
- Toutes les méthodes d'une interface sont par défaut abstract et public

Les threads

Processus

- Un processus est une unité d'exécution faisant partie d'un programme.
- Fonctionne de façon autonome et parallèlement à d'autres processus.
- Sur une machine mono processeur,
  - chaque unité se voit attribuer des intervalles de temps au cours desquels elle a le droit d'utiliser le processeur pour accomplir ses traitements.

Processus

- Le système alloue de la mémoire pour chaque processus :
  - segment de code (instructions),
  - segment de données (allocations mémoire),
  - segment de pile (variables locales).
- le système associe à chaque processus
  - identificateur
  - priorités,
  - droits d'accès…

Etats des processus

- **Nouveau** (new): le processus est créé mais pas encore démarré
- **Exécutable** (runnable): la méthode de démarrage a été appelée
- **En exécution** (running) : le processus s'exécute sur un processeur du système
- **En attente** (waiting) : le processus est prêt à s'exécuter, mais n'a pas le processeur (occupé par un autre processus en exécution) ;
- **bloqué** (blocked): il manque une ressource (en plus du processeur) au processus pour qu'il puisse s'exécuter ou est arrêté à cause par exemple d'un timer.
- **Terminé** (terminated): il termine (fin normale de l'exécution ou levée d'une exception)

Les threads

- Appelé aussi processus léger
- Un thread est un processus à l'intérieur d'un autre processus
- Les ressources allouées à un processus vont être partagées entre les threads qui le composent
- Chaque thread possède son propre environnement d'exécution (valeurs des registres du processeur) ainsi qu'une pile (variables locales).

Les threads Java

- L'utilisation/comportement des threads changent d'un OS à un autre
- Java définit son propre concept de thread:
  - La JVM permet l'exécution concurente de plusieurs threads
  - Chaque thread possède une priorité. Les threads de plus haute priorité s'exécutent en premier
  - La JVM démarre avec l'exécution du thread correspondant à la méthode main

Codage

- Un thread est un objet Java qui
  - est une instance d'une classe qui hérite de la classe Thread,
  - ou qui implémente l'interface Runnable.

Par héritage

- Déclaration
  - class MonThread extends Thread

    {

    public void run()

    { // traitement parallèle . . . }

    }
- Création
  - MonThread p = new MonThread ();
  - p.start();

Par implémentation

- Déclaration
  - class MonRunnable implements Runnable

```
{
public void run()
{ // traitement . . . }
}
```

- Création
  - MonRunnable p = new MonRunnable ();

```
Thread q = new Thread(p);
 q.start();
```

La classe Thread

- Fait partie du package java.lang
- Un constructeur sans arguments
- D'autres constructeurs pouvant considérer les arguments
  - Un nom: le nom du Thread (par défaut, le préfixe Thread- suivi par un numéro incrémenté automatiquement)
  - Un objet qui implémente l'interface Runnable: contient les traitements
  - Un group: le groupe auquel sera rattaché le Thread

Quelques méthodes

void destroy()	met fin brutalement au thread
String getName()	Renvoie le nom du thread
long getId()	renvoie un numéro donnant un identifiant au thread
int getPriority()	renvoie la priorité du thread
boolean isAlive()	renvoie un booléen qui indique si le thread est actif ou non
boolean isInterrupted()	renvoie un booléen qui indique si le thread a été interrompu
void join()	attend la fin de l'exécution du thread
void join(long m)	attend au max m millisecondes la fin de l'exécution du thread
void sleep(long m)	mettre le thread en attente durant le temps exprimé en millisecondes fourni en paramètre.
void start()	démarrer le thread et exécuter la méthode run()
…	…

Exemple

```
public class T extends Thread
{
public void run()
{
int i;
for(i=0;i<5;i++)
{
System.out.println("je suis vivant");
try{
this.sleep(1);
}
catch(Exception e)
{}}}
public static void main(String args[]) throws Exception
{
T t=new T();
t.start();
while(t.isAlive())
{
System.out.println("le processus est vivant");
}}}
```

Partager les variables entre Threads

- Les threads d'un même processus partagent le même espace mémoire.
- Chaque instance de la classe thread possède ses propres variables (attributs).
- Pour partager une variable entre plusieurs threads, on a souvent recours à une variable de classe (par définition partagée par toutes les instances de cette classe)

Exemple

```
public class Partager extends Thread
{               private static String nomGlobal= "";
                private String nomLocal;
                Partager ( String s ) { nomLocal = s; }
                public void run()
                {
                nomGlobal = nomGlobal + nomLocal;
                System.out.println(nomGlobal);
                }
                public static void main(String args[])
                {
                Thread T1 = new Partager( "Toto" );
                Thread T2 = new Partager( "Titi" );
                T1.start(); T2.start();
                System.out.println( nomGlobal );
                System.out.println( nomGlobal );
                System.out.println( nomGlobal );
                System.out.println( nomGlobal ); } }
```

Synchronisation: motivation

- Exemple

```
public class Tableau
{
int tab[];
int indice;
public Tableau(int[] T)
{
tab=T;
indice=0;
}

public void lire()
{
System.out.println("lu: "+tab[indice]);
indice++;
}
}
```

```
public class Lecteur extends Thread
{
public Tableau t;
public Lecteur(Tableau Tab)
{
t=Tab;
}
public void run()
{
t.lire();t.lire();t.lire();t.lire();t.lire();
}

public static void main(String args[])
{
int tabEntiers[]=new int[10];
for(int i=0;i<10;i++)
{
tabEntiers[i]=i;
}
Tableau tab=new Tableau(tabEntiers);
Lecteur L1=new Lecteur(tab);
Lecteur L2=new Lecteur(tab);
L1.start();
L2.start();}}
```

Exclusion mutuelle et section critique (arrêt)

- Utilisation de sections synchronisées:

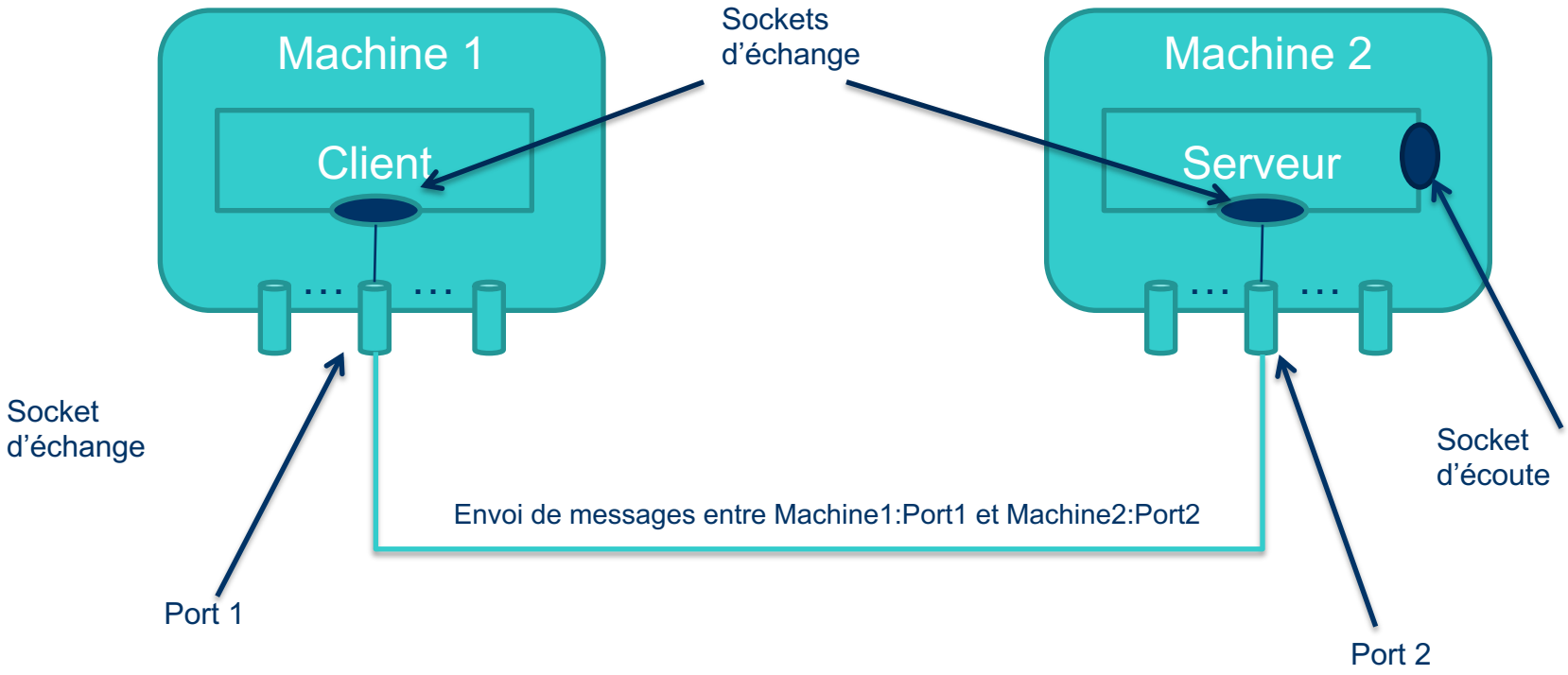
```
public class Tableau
{
…
public void lire()
{
synchronized(this)
      {
         System.out.println("lu: "+tab[indice]);
          indice++;
      }
}
```

```
public class Tableau
{
…
public synchronized void lire()
{
System.out.println("lu: "+tab[indice]);
indice++;
}
}
```

- Interdit l'exécution de la section critique sur le même objet de manière entrelacée

La communication par Sockets

Architecture



Le package

- Le package utilisé pour l'implantation de la communication par sockets: java.net
- Comprend les classes
  - java.net.InetAddress: permet de manipuler les adresses IP
  - java.net.SocketServer: permet de programmer l'interface côté serveur (sockets d'écoute)
  - java.net.Socket: permet de programmer l'interface côté client (sockets connectées)

La classe InetAddress (1/2)

- 3 méthodes statiques pour créer des objets adresse IP
  - public static InetAddress getLocalHost() throws UnknownHostException
    - renvoie l'adresse IP du site local d'appel.
  - public static InetAddress getByName(String host) throws UnknownHostException
    - construit un nouvel objet InetAddress à partir d'un nom textuel de site.
    - Le nom du site est donné sous forme symbolique (www.uic.ac.ma) ou sous forme numérique (147.127.18.03).
  - public static InetAddress[] getAllByName(String host) throws UnknownHostException
    - permet d'obtenir les différentes adresses IP d'un site.

La classe InetAddress (2/2)

- Trois méthodes pour obtenir les informations sur l'objet adresse IP
  - public String getHostName()
    - obtient le nom complet correspondant à l'adresse IP
  - public String getHostAddress()
    - obtient l'adresse IP sous forme %d.%d.%d.%d
  - public byte[] getAddress()
    - obtient l'adresse IP sous forme d'un tableau d'octets

La classe ServerSocket (1/3)

- Les constructeurs
  - public ServerSocket()
    - Créée un objet socket d'écoute non liée
  - public ServerSocket(int p) throws IOException
    - Crée un objet socket à l'écoute du port p

La classe ServerSocket (2/3)

- Les méthodes
  - public Socket accept() throws IOException
    - Acceptation de la connection d'un client
    - Opération bloquante
    - Par défaut, le temps d'attente est infini
  - public void setSoTimeout(int timeout) throws SocketException
    - L'argument est en millisecondes
    - Définit un délai de garde
    - À l'expiration, l'exception java.io.InterruptedIOException est levée
  - public void close()
    - Ferme la socket d'écoute

La classe ServerSocket (3/3)

- Les getters
  - public InetAddress getInetAddress()
    - Permet de récupérer l'objet adresse IP
  - public int getLocalPort()
    - Permet de récupérer le port d'écoute

La classe Socket (1/3)

- Utilisée pour la programmation des sockets connectées côté client et serveur.
- Création
  - Côté Serveur: résultat de l'appel de la méthode accept
  - Côté Client: par l'appel des constructeurs
    - public Socket(String host, int port) throws UnknownHostException, IOException
      - Ouvre une socket sur une machine et un port côté serveur. Le choix côté client n'est pas spécifié.
    - public Socket(InetAddress address, int port) throws IOException
      - Utilise l'objet InetAddress au lieu d'une chaine de caractères
    - public Socket(String host, int port, InetAddress localAddr, int localPort) throws UnknownHostException, IOException
      - Spécifie une adresse et un port côté Client
    - public Socket(InetAddress addr, int port, InetAddress localAddr, int localPort) throws IOException

La classe Socket (2/3)

- Méthodes
  - public OutputStream getOutputStream() throws IOException
    - Ouvre un flux d'écriture sur la socket
    - Permet de construire un objet PrintWriter
  - public InputStream getInputStream() throws IOException
    - Ouvre un flux de lecture sur la socket
    - Permet de construire un objet BufferedReader
    - L'opération de lecture est bloquante
  - Possibilité de définir un délai de garde
    - public void setSoTimeout(int timeout) throws SocketException
  - public void close()
    - Ferme la socket et libère les ressources

La classe Socket (3/3)

- Les getters
  - public InetAddress getInetAddress()
    - L'adresse IP distante
  - public InetAddress getLocalAddress()
    - L'adresse IP locale
  - public int getPort()
    - Le port distant
  - public int getLocalPort()
    - Le port local

Ecriture du code: côté serveur

- Créer une socket d'écoute
  - ServerSocket SS= new ServerSocket(NumeroPort);
- Récupérer la socket d'échange
  - Socket S=SS.accept();
- Ouvrir un flux de lecture sur la socket
  - BufferedReader BR = new BufferedReader(new InputStreamReader(S.getInputStream()));
- Ouvrir un flux d'écriture sur la socket
  - PrintWriter PW = new PrintWriter(S.getOutputStream());
- Réaliser les traitements
  - PW.println("toto");
  - PW.flush();
  - String message=BR.readLine();
- Fermer les flux de lecture et d'écriture
- Fermer la socket d'échange

Ecriture du code: côté Client

- Créer la socket d'échange
  - Socket S=new Socket(AdresseServeur,port);
- Ouvrir un flux de lecture sur la socket
  - BufferedReader BR = new BufferedReader(new InputStreamReader(S.getInputStream()));
- Ouvrir un flux d'écriture sur la socket
  - PrintWriter PW = new PrintWriter(S.getOutputStream());
- Réaliser les traitements
- Fermer les flux de lecture et d'écriture
- Fermer la socket d'échange

RMI:
Remote Method Invocation

Remote Method Invocation

- Technologie pour la mise en œuvre d'objets distribués
- Le but de RMI est de permettre l'appel, l'exécution et le renvoi du résultat d'une méthode exécutée dans une machine virtuelle différente de celle de l'objet l'appelant
- S'appuie sur le stub côté client pour implémenter la communication distante
- Package: java.rmi

Les deux parties

- Une application RMI est constituée généralement de deux programmes:
  - Partie Serveur:
    - Crée des objets distants
    - Crée des références sur ces objets pour les rendre accessibles
    - Attend les requêtes des clients: appels distants sur des méthodes sur ces objets
  - Partie Client:
    - Obtient une ou plusieurs références sur des objets distants localisés sur un ou plusieurs serveurs
    - Effectue des appels distants sur les méthodes des objets

Interfaces, objets et méthodes distantes

- Comme n'importe quelle application Java, une application distribuée RMI est constituée d'interfaces et de classes
  - Les interfaces déclarent des méthodes
  - Les classes implémentent ces méthodes et d'autres méthodes additionnelles
- Les objets possédant des méthodes pouvant être appelées à travers plusieurs JVMs sont appelés objets distants.
  - Ils implémentent une interface distante (remote interface) possédant les caractéristiques suivantes:
    - Hérite de java.rmi.Remote
    - Chaque méthode de l'interface peut lever l'exception: java.rmi.RemoteException

Fonctionnement de l'invocation distante dans RMI

- RMI traite un objet distant de manière différente par rapport à un objet local
  - Au lieu de créer une copie de l'objet sur l'autre JVM (côté client), RMI utilise un stub pour chaque objet distant:
    - Le stub joue le rôle d'un représentant local (ou proxy) pour l'objet distant
    - Le client invoque la méthode sur le stub local
    - Le stub transporte l'appel de méthode vers l'objet distant
- Le stub d'un objet distant implémente la même interface distante que l'objet distant
  - Permet de caster toutes les méthodes
  - Seules les méthodes déclarées dans l'interface peuvent être invoquées à distance

Etapes de création des applications RMI

- Le développement d'une application RMI correspond à 4 étapes
  - Design et implémentation des composants de l'application distribuée
  - Compilation des sources
  - Publication et mise à disposition des objets distants
  - Démarrage de l'application

Design et implémentation

- Définir l'architecture de l'application
  - Quels objets sont locaux et lesquels sont distants
  - Définir les interfaces distantes pour les composants distants
    - Spécifier les méthodes invocables à distance par les clients (types des paramètres et type de retour)
- Implémentation des objets distants:
  - Les objets distants
    - Doivent implémenter une ou plusieurs interfaces distantes
    - Peuvent implémenter d'autres interfaces et méthodes qui ne peuvent être appelées que localement
- Implémentation des clients:
  - Les clients qui utilisent les objets distants peuvent être implémentés à n'importe quel moment après la définition des interfaces

Compilation des sources

- Génération des codes exécutables:
  - Compiler les codes sources (serveurs , clients, classes d'implémentation…)
  - => javac
- Génération du stub
  - Compiler l'interface
  - => rmic

Rendre les objets accessibles

- Enregistrement sur le registre rmi
  - Spécification d'une localisation
  - Attribution d'un nom
  - Méthode bind (rebind) de la classe java.rmi.Naming

Lancement de l’application

- Exécution:
  - Du registre d’objets RMI
  - Du Serveur
  - Du client

Récapitulation

- Côté serveur:
  - Définition de l'interface d'accès distant
  - Implémentation de la classe de l'objet distant
  - Implémentation du serveur qui crée l'objet distant et l'enregistre sur le registre de nommage RMI (RMI registry)
- Côté client: implémentation comprenant
  - L'obtention d'une référence sur l'objet distant à travers le registre de nommage RMI
  - La réalisation des appels de méthodes distantes en utilisant cette référence

Exemple simple

Un exemple simple: L'objet Hello

- L'application comprend
  - L'interface:
    - HelloInt: l'interface de l'objet distant
  - Les classes:
    - Hello: implémente l'objet
    - Serveur: la partie serveur de l'application
    - Client: la partie client de l'application

L’interface

- Doit hériter de l’interface java.rmi.Remote
- Publie les méthodes accessibles par appels distants
- Dans un fichier HelloInt.java:

```
package hello;
import java.rmi.*;
public interface HelloInt extends Remote
{
public void SayIt() throws RemoteException;
}
```

L’implémentation

- Implémente le code de l’objet distant
- Implémente toutes les méthodes déclarées dans l’interface
- Hérite de la classe UnicastRemoteObject
- Toutes les méthodes y compris le constructeur lèvent l’exception RemoteException

La classe Hello.java

```
package hello;
import java.rmi.*;
import java.rmi.server.*;
public class Hello extends UnicastRemoteObject implements
    HelloInt {
            public Hello() throws RemoteException {
            super();
            }
public void SayIt()throws RemoteException {
System.out.println("bonjour");
}
}
```

Le serveur

- Crée l'objet distant
- L'enregistre sur le serveur de nommage RMI
- Dans un fichier Serveur.java

```
import java.rmi.*;
import java.rmi.registry.*;
import hello.*;
public class Serveur{
public static void main(String[] args) throws Exception {
// création de l'objet
Hello h = new Hello();
```

Le serveur

- L'enregistrement d'un objet se fait à travers la méthode rebind
- Prend en paramètre:
  - L'objet
  - Sa localisation: port et nom attribué
- Introduire le reste du code du serveur:

```
System.out.println("Je vais enregistrer mon objet");
Naming.bind("rmi://localhost:1099/hello",h);
System.out.println("C'est fait!");
}}
```

Le client

- La première étape est d'obtenir une référence sur l'objet distant:
  - Utiliser la méthode statique lookup de la classe naming
  - Retourne le stub en faisant une recherche sur le nom de l'objet référencé dans le registre de nommage
  - Retourne l'exception NotBoundException si la recherche échoue

Le code du client

```
import hello.*;
import java.rmi.*;
public class Client{
public static void main(String[] args)
{
    try {
        Remote remote_h = Naming.lookup("rmi://localhost:1099/hello");
        if (remote_h instanceof HelloInt)
            {
                ((HelloInt) remote_h).SayIt();
            }
    }
catch (Exception e) {
}
}}
```

Structuration de l'application

- Côté Serveur
  - Interface et implémentation de l'objet distant
  - Implémentation du serveur
  - Compilation du serveur et de l'objet distant
    - javac *.java
  - Génération du stub
    - rmic hello.Hello
- Côté Client
  - Interface et Stub
  - Implémentation du client
  - Compilation du client

Exécution

- Dans le répertoire MON_PREMIER_RMI
  - Lancement du registre de nommage
    - java sun.rmi.registry.RegistryImpl 1099
  - Lancement du serveur
    - java Serveur
- Lancement du client
  - java Client

Les différents sens de la communication

- Du Client vers le Servant
- Du Servant vers le Client
- Du Serveur vers le Servant
- Du Serveur vers le Client

A vous de jouer!

- Afficher un message de retour chez le client
- En considérant plusieurs clients et plusieurs Serveurs:
  - Passer le nom du client au serveur
  - Passer le nom du serveur au client
- En créant un autre objet Bonjour, et un autre ServeurBis, construire une architecture en séquence où le client appelle Serveur qui appelle ServeurBis