

SYSTÈMES RÉPARTIS



Joseph NDONG
Faculté des Sciences et Techniques
Département de Mathématiques et Informatique

Joseph NDONG UCAD DAKAR
SENEGAL



PLAN DU COURS

Module 1: Sérialisation

Module 2: Programmation réseau avec les Socket

Module 3: RMI

Module 4: L'introspection

Module 5: JavaBeans



Module 1

La sérialisation

La programmation d'applications réparties en Java utilise des notions qui ne sont pas propres aux applications réparties, il s'agit de la sérialisation que nous allons voir dans ce chapitre.

La sérialisation est un mécanisme fondamental en Java qui permet de gérer la **persistance des objets**.

Elle permet de transférer un objet vers un support binaire.

Les objets sont mis dans une forme où ils pourront être reconstitués exactement à l'identique.

Les objets pourront alors être stockés sur disque dur ou véhiculés à travers le réseau pour être recréés dans une autre JVM. C'est le procédé qu'utilise RMI.

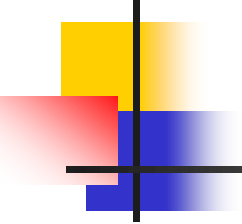
La **désérialisation** de l'objet doit se faire avec la **même classe** qui a été utilisée pour la **sérialisation**.



Classes et interfaces

La sérialisation définit l'*interface* **Serializable** et les *classes* **ObjectOutputStream** et **ObjectInputStream**.

Remarques



Tous les objets ne sont pas sérialisables: soit parce qu'ils sont liés à des primitives du système, soit parce qu'ils sont liés à un contexte d'exécution en mémoire tels que les threads. Les objets fichiers (**File**) sont devenus sérialisables dans les versions récentes de Java.

Si vous utilisez l'outil Java **Serialver** sur la classe **java.awt.Graphics**, on vous dira que la classe n'est pas sérialisable.

```
| serialver java.awt.Graphics  
| Class java.awt.Graphics is not Serializable.
```

Par contre la classe **File** est sérialisable

```
| serialver java.io.File  
| java.io.File: static final long serialVersionUID = 301077366599181567L;
```



L' interface **java.io.Serializable**

L' interface **Serializable** ne *définit aucune méthode* mais permet simplement de préciser une classe comme pouvant être sérialisée.

Toute classe qui gère des objets sérialisés doit implémenter cette interface ou une de ses classes mères doit l'implémenter.

Si l'on tente de sérialiser un objet d'une classe qui n'implémente pas l'interface **Serializable**, une exception de type **NotSerializableException** est levée.

*Un objet sérialisé, sérialise à son tour chaque variable (champ) de sa classe.
On prendra alors garde sur les objets non sérialisables.*

Voici un exemple de classe Serializable:



Exemple de classe Serializable

```
public class Personne implements java.io.Serializable {  
    private String nom ;  
    private String prenom ;  
    private int taille;  
    public Personne(String nom, String prenom, int taille) {  
        this.nom = nom;  
        this.taille = taille;  
        this.prenom = prenom;  
    }  
    public String getNom () { return nom; }  
    public void setNom (String nom) { this.nom = nom; }  
    public int getTaille () { return taille; }  
    public void setTaille (int taille) { this.taille = taille; }  
    public String getPrenom () { return prenom; }  
    public void setPrenom (String prenom) { this.prenom = prenom; }  
}
```

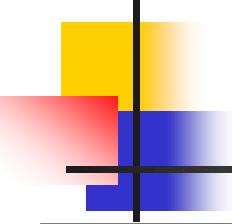
La classe ObjectOutputStream

Cette classe permet de sérialiser un objet.

```
package serialiser;
import java.io.*;
public class SerializerPersonne {
    /*cette methode permet de serialiser n importe quel individu*/
    public File serializerUnePersonne (Personne p){
        try { File fichier = new File ("D:\\ all_ndong\\personne.txt");
            FileOutputStream fic = new FileOutputStream (fichier);
            ObjectOutputStream oos = new ObjectOutputStream (fic);
            oos.writeObject (p) ;
            oos.flush () ; // on vide le contenu du flux
            oos.close () ;
            return fichier;
        }

        catch (IOException e) { return null;}
    }
}
```


La classe **ObjectOutputStream**



Dans la méthode **serialiserUnePersonne**, on construit d'abord un objet *File* destiné à contenir l'objet sérialisé (en fait les attributs de cet objet), ensuite avec la classe *FileOutputStream*, on associe un flux en écriture au fichier de destination et enfin on sérialise l'objet dans sa globalité en instanciant un objet de la classe *ObjectOutputStream*.

L'écriture de l'objet dans le flux est réalisée par la méthode **writeObject**.

Lors de ces opérations, une exception de type *IOException* peut être levée si un quelconque problème survient avec le fichier.

La classe ObjectInputStream

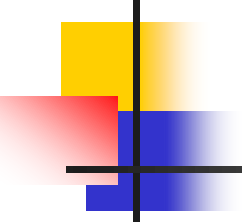
Cette classe permet de désérialiser un objet.

```
package serialiser;
import java.io.*;
public class DeserializerPersonne {
    /*cette methode permet de deserialiser n importe quel individu*/
    public void deserializerUnePersonne (File f){
        try {
            FileInputStream fic = new FileInputStream (f);
            ObjectInputStream ois = new ObjectInputStream (fic);
            Personne p = (Personne) ois.readObject();
            System.out.println ("la personne qui était serialisée est :");
            System.out.println (p.getPrenom ());
            System.out.println (p.getNom());
            System.out.println (p.getTaille ());
        }
        catch (IOException e) { }
        catch (ClassNotFoundException ez) { }
    }
}
```

1

1

La classe `ObjectInputStream`



Dans la méthode *deserializerUnePersonne* on crée un objet de type `FileInputStream` qui représente un flux en lecture sur le fichier contenant l'objet sérialisé. Ensuite on crée un objet `ObjectInputStream` qui permet grâce à sa méthode *readObject* de récupérer l'objet sérialisé.

Un cast est nécessaire pour obtenir le type réel de l'objet sérialisé.

Si la classe a changé entre le moment où elle a été sérialisée et le moment où elle est désérialisée, une exception de type `InvalidClassException` est levée.

Une exception de type `StreamCorruptedException` peut être levée si le fichier a été corrompu par exemple en le modifiant avec un éditeur.

Une exception de type `ClassNotFoundException` peut être levée si l'objet est transtypé vers une classe qui n'existe plus au moment de l'exécution.



Une classe de test

```
package serialiser;  
import java.io.*;  
public class TestSerializable {  
    public static void main (String[] args) {  
        Personne p=new Personne ("Ndong","Joseph",180);  
        File f= new SerializerPersonne(). serializerUnePersonne(p);  
        new DeserializerPersonne(). deserializerUnePersonne(f);  
    }  
}
```

la personne qui était serialisée est :
Joseph
Ndong
180

Le mot clé **transient**

Le contenu des attributs sont visibles dans le flux dans lequel est sérialisé l'objet.

Il est ainsi possible pour toute personne ayant accès au flux de voir le contenu de chaque attribut même si ceux-ci sont **private**. Ceci peut poser des problèmes de sécurité surtout si les données sont sensibles et transmises à travers un réseau.

Java introduit le mot clé **transient qui précise que l'attribut qu'il qualifie ne doit pas être inclus dans un processus de sérialisation et donc de désérialisation.**

Lors de la désérialisation, les champs **transient** sont initialisés avec la valeur **null**.

Ceci peut poser des problèmes à l'objet qui doit gérer cet état en provoquant des exceptions de type *NullPointerException*.

Si on déclare l'attribut nom avec le mot clé **transient, à la désérialisation la sortie du programme précédent sera:**

la personne qui était sérialisée est :

Joseph

null

180

L'interface **java.io.Externalizable**

Pour créer un objet persistant sans utiliser le mécanisme de sérialisation par défaut (ie sans utiliser l'interface **Serializable**), vous pouvez effectuer une sérialisation personnalisée en implémentant l'interface **Externalizable**.

Cette interface possède deux méthodes, **writeExternal** et **readExternal** que vous aurez à redéfinir pour l'écriture et la lecture des données.

Par défaut, la sérialisation d'un objet qui implémente cette interface **ne prend en compte aucun attribut (champ) de l'objet**. Par conséquent l'usage du mot clé **transient** est inutile.

En redéfinissant la méthode *readExternal*, vous utiliserez les méthodes de **DataInput** pour les types primitifs et **readObject** pour les valeurs de type String, Object et arrays (tableaux).

De même pour *writeExternal*, utilisez les méthodes de **DataOutput** pour les types primitifs et **writeObject** pour les String, Object et arrays.

Exemple de classe Externalizable (1/3)

(repreons l'exemple de la classe *Personne* avec cette interface dans une version que nous appelons *Personne2*)

```
public class Personne2 implements java.io.Externalizable {  
    private String nom ;  
    private String prenom ;  
    private int taille;  
    public Personne2 (String nom, String prenom, int taille) {  
        this.nom = nom;  
        this.taille = taille;  
        this.prenom = prenom;  
    }  
    public String getNom2 () { return nom; }  
    public void setNom2 (String nom) { this.nom = nom; }  
    public int getTaille2 () { return taille; }  
    public void setTaille2 (int taille) { this.taille = taille; }  
    public String getPrenom2 () { return prenom; }  
    public void setPrenom2 (String prenom) { this.prenom = prenom; }  
}
```

Exemple de classe Externalizable (2/3)

*/*redéfinition de la méthode readExternal*/*

```
public void readExternal (ObjectInput e) {
```

```
    try {
        System.out.println("starting reading...");
        System.out.println (e.readObject ( ) );
        System.out.println (e.readObject ( ));
        System.out.println (e.readInt ( ) );
    }
    catch (Exception enn) { System.out.println ("Erreur reading...");
        enn.printStackTrace ( );
    }
```

} //fin de la methode readExternal

Exemple de classe Externalizable (3/3)

*/*redéfinition de la méthode writeExternal*/*

```
public void writeExternal (ObjectOutput e) {
```

```
    try { System.out.println(" starting writing...");
        e.writeObject ( this.getNom2( ) );
        e.writeObject (this.getPrenom2( ) );
        e.writeInt ( this.getTaille2( ) );
        e.flush() ;
        e.close() ;
        System.out.println("finishing writing !!!");
    }
    catch (Exception enn) { System.out.println ("Erreur reading...");
        enn.printStackTrace ( );
    }
}
```

} //fin de la methode writeExternal

} //fin de la classe Personne2

Joseph NDONG UCAD DAKAR

SENEGAL

Une classe pour tester

writing...
finishing writing!!!
starting reading...

Sene
Pierre
178

```
import java.io.*;
public class TestPersonne2 {
    public static void main (String args [ ] ){
        Personne2 c = new Personne2 ("Sene","Pierre",178);

        try {
            File f = new File ("c:\\alldong\\individuX.txt");
            FileOutputStream fos = new FileOutputStream ( f );
            java.io.ObjectOutput oos = new ObjectOutputStream (fos);
            c.writeExternal (oos); //objet serialisé
            FileInputStream fis = new FileInputStream ( f );
            ObjectInputStream ois = new ObjectInputStream ( fis );
            c.readExternal ( ois ); //objet désérialisé
        }
        catch (Exception enn) {System.out.println ("Erreur main"); }
    }
}
```

Joseph NDONG UCAD DAKAR
SENEGAL



Exercice

Realiser une classe *CompteBank* qui manipule des comptes bancaires et sérialiser les objets issus de l'instanciation de cette classe.

Un compte pourra être identifié par le numéro, le solde, le prénom et le nom du titulaire, le découvert.

La sérialisation s'effectuera dans un fichier sur disque.

Créer une classe de désérialisation et afficher le solde des comptes.

Rajoutez une classe Client (un client étant identifié par N°CIN, Prénom, Nom).

Créez des objets que vous sérialisez dans le même fichier qui a servi pour la sérialisation des objets comptes.

Créer maintenant une classe qui désérialise tous les objets du dit fichier.

Afficher par exemple le nom du client et le solde du compte.

FIN



Module 2

Programmation réseau: Socket

Depuis son origine, Java fournit plusieurs classes et interfaces destinées à faciliter l'utilisation du réseau par programmation.

La couche **transport** du modèle OSI (Open System Interconnect) est implémentée dans les protocoles UDP ou TCP. **Ils permettent la communication entre des applications sur des machines distantes.** La notion de service permet à une même machine d'assurer plusieurs *communications simultanément*.

En Java, ce service est établi à l'aide des **Sockets**.

Le système des sockets est le moyen de communication *inter processus* développé pour l'Unix Berkeley (BSD). Il est actuellement implémenté sur tous les systèmes d'exploitation utilisant TCP/IP.

Un socket est le point de communication par lequel un thread peut émettre ou recevoir des informations et ainsi elle permet la communication entre deux applications à travers le réseau.



Les sockets

Grâce aux **sockets**, une connexion réseau peut désormais être traitée comme un flux d'octets ordinaires, dans lequel le programmeur peut lire et écrire à merveille.

La communication se fait sur un **port** particulier de la machine. Le port est une entité logique qui permet d'associer un service particulier à une connexion. Un port est identifié par un entier de 1 à 65535. Par convention les 1024 premiers sont réservés pour des services standards (80 : HTTP, 21 : FTP, 25: SMTP, ...)

Java prend en charge deux protocoles : TCP et UDP.

Les classes et interfaces utiles au développement réseau sont regroupées dans le package **java.net .**



Les sockets

un socket assure sept (7) opérations élémentaires:

- se connecter à une machine distante (en prévision pour l'envoi et la réception)
- envoyer des données;
- recevoir des données;
- clore une connexion;
- s'attacher à un port;
- attendre les demandes de connexion provenant de machines distantes;
- accepter les demandes de connexions sur le port local.

Les quatre (4) premières sont prises en charge par les méthodes de la classe **Socket**, sollicitée par les *clients* comme les *serveurs*.

Les trois (3) dernières, requises par les *serveurs* en attente de connexion client, sont implantées par la classe **ServerSocket**.



Les sockets

Voici un aperçu de la mise en œuvre d'un socket client:

- 1. Le socket est créé au moyen d'un constructeur `Socket ()`.**
- 2. Le nouveau socket tente de se connecter à un hôte distant.**
- 3. Une fois la connexion établie, le système local et celui distant mettent en place un *canal de communication* à partir du socket et l'utilisent pour échanger des données. Cette connexion simultanée dans les deux sens est dite *full-duplex*.
Après la procédure d'authentification appropriée, l'échange des données peut commencer. La nature des données échangées dépend du protocole employé: un serveur FTP, par exemple, ne doit pas recevoir les mêmes commandes qu'un serveur HTTP.**
- 4. Lorsque les échanges sont terminés, un des deux interlocuteurs clôt la connexion. Certains protocoles, comme HTTP, exigent de **clôre celle-ci après chaque requête** honorée tandis que d'autres, comme FTP, autorisent le traitement de plusieurs requêtes au cours d'une même connexion.**

La classe **java.net.InetAddress**

(cette section sur la classe *InetAddress* et les *URLs* aident aussi au développement des *Applets java*)

Rappel sur les adresses Internet

Une adresse internet permet d'identifier de *façon unique une machine sur un réseau*. Cette adresse pour le protocole I.P est sous la forme de **quatre** octets séparés chacun par un point. Chacun de ces octets appartient à une classe selon l'étendue du réseau. Pour faciliter la compréhension humaine, un serveur particulier appelé DNS (**D**omaine **N**ame **S**ervice) est capable d'associer un nom à une adresse IP.

Un objet de la classe *InetAddress* représente *une adresse Internet*. Elle contient des méthodes pour lire une adresse, la comparer avec une autre ou la convertir en chaîne de caractères. **Elle ne possède pas de constructeur : il faut utiliser certaines méthodes statiques de la classe pour obtenir une instance de cette classe (cf diapo suivante...).**

La classe `java.net.InetAddress`

Cette classe ne contient que deux champs: le champ `hostName` qui doit contenir une chaîne de caractère stockant ainsi le nom de l'hôte (`www.ucad.sn`, par exemple) et le champ `address` qui doit contenir un entier renfermant les 32 bits de l'adresse numérique IP.

A noter qu'aucun de ces deux champs n'est public ie accessible directement.

Création d'objets **java.net.InetAddress** (1/5)

La classe `InetAddress` ne possède aucun constructeur. Il existe cependant *trois méthodes statiques* qui, en échange des informations indispensables, renvoient des objets `InetAddress` correctement initialisés. Il s'agit:

```
public static InetAddress getByName (String nom_hote)
public static InetAddress [ ] getAllByName (String nom_hote)
public static InetAddress getLocalHost ()
```

Voyons maintenant dans quels cas utiliser ces méthodes.

Création d'objets **java.net.InetAddress** (2/5)

Vous aurez le plus souvent recours à **InetAddress.getByName (...)** qui s'appuie sur le DNS pour rechercher l'adresse IP de l'hôte dont le nom est passé en paramètre.

Voici un exemple:

```
InetAddress adr = InetAddress.getByName ("www.ucad.sn");
```

Lorsque la recherche n'aboutit pas, la méthode **InetAddress.getByName ()** retourne une **UnknownHostException**. Ainsi il faut inclure l'appel dans un bloc try:

```
try {  
    InetAddress adr = InetAddress.getByName ("www.ucad.sn");  
    System.out.println (adr) ;  
}  
catch (UnknownHostException e) {  
    System.out.println ("www. ucad.sn:adresse non trouvée ") ;  
}
```

Création d'objets **java.net.InetAddress** (3/5)

Certains ordinateurs ont plus d'une adresse Internet. Pour rechercher celles-ci, on fera appel à la méthode **getAllByName (...)** qui renvoie un tableau contenant l'ensemble des *InetAddress* d'un hôte donné. On l'utilise comme ceci:

```
InetAddress [ ] adr = InetAddress.getAllByName ("www.ucad.sn");
```

Cette méthode est susceptible de générer une **UnknownHostException**. Ainsi comme pour *getByName ()*, il faut l'envelopper dans un bloc **try**.

```
public class AllAddresses { // programme affichant les adresses de www.ucad.sn  
    public static void main (String args [ ]){  
        try {  
            InetAddress [ ] adr = InetAddress.getAllByName ("www.ucad.sn");  
            for (int i = 0; i < adr.length; i++)  
                System.out.println (adr [i] );  
        }  
        catch (UnknownHostException e) { }    } }
```

Création d'objets **java.net.InetAddress** (4/5)

La méthode **getLocalHost** () renvoie l'adresse de la machine sur laquelle elle s'exécute (il s'agit de la machine locale). Comme pour les deux méthodes précédentes, une recherche infructueuse se traduit par la levée d'une **UnknownHostException**.

```
public class LocalHostAddress { // programme affichant l'adresses de cette machine
    public static void main (String args [ ]){
    try {
        InetAddress adr = InetAddress.getLocalHost ( );

        System.out.println ("adresse de cette machine:" +adr );
    }
    catch (UnknownHostException e) {
        System.out.println ("Impossible de trouver l'adresse de cette machine:" );}
    } // fin de la méthode main

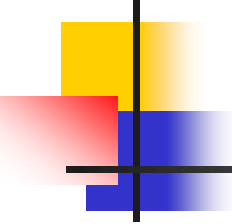
} // fin de la classe
```

Création d'objets **java.net.InetAddress** (5/5)



Le paquet **java.net** propose en plus quatre autres méthodes capables d'instancier des objets `InetAddress`: **`Socket.getInetAddress ()`**, **`ServerSocket.getInetAddress ()`**, **`SocketImpl.getInetAddress ()`** et **`DatagramPacket.getAddress ()`**, que nous étudierons dans les chapitres ultérieurs.

Accès aux champs de la classe InetAddress



L'accès aux champs de la classe `InetAddress` est réservé aux seules classes de **java.net**. Les méthodes `getHostName ()` et `getAddress ()` permettent aux classes issues d'autres paquets de lire le contenu des champs d'un objet `InetAddress`, mais l'absence des méthodes correspondantes `setHostName ()` et `setAddress ()` interdit à ces dernières de modifier la valeur de ces champs.

De cette façon, Java garantit que le nom d'hôte et l'adresse IP concordent et désignent de surcroît un hôte répertorié.

Accès aux champs de la classe InetAddress



```
public String getHostName ( )
```

La méthode *getHostName ()* renvoie une chaîne contenant le nom d'hôte de l'objet *InetAddress*, ou bien l'adresse numérique si la machine ne possède pas de dénomination littérale.

```
public class LocalHostName { // programme affichant le nom de cette machine
    public static void main (String args [ ]){
        try {
            InetAddress adr = InetAddress.getLocalHost ( );

            System.out.println ("Salut. Je m'appelle:" +adr. getHostName ( ));
        }
        catch (UnknownHostException e) {
            System.out.println ("Désolé. Je ne connais pas mon nom!");
        } // fin de la méthode main
    } // fin de la classe
```


Accès aux champs de la classe InetAddress



```
public byte [ ] getAddress ( )
```

La recherche de l'adresse IP d'une machine n'est que rarement nécessaire. Pour l'occasion, on se servira de `getAddress ()`, qui retourne cette information sous la forme d'un tableau d'octets rangés dans le même ordre que l'adresse numérique.

La manipulation d'octets non signés pose certains problèmes en Java qui ne prédéfinit pas ce type de données et traite les octets supérieurs à 127 comme des nombres négatifs. Aussi, avant de manipuler les octets renvoyés par `getAddress ()`, il convient de les Convertir en entiers et d'effectuer certains ajustements:

```
int octetNonSigne = monOctet < 0 ? monOctet + 256 : monOctet ;
```

Voici un exemple de recherche de l'adresse IP de la machine locale.

Accès aux champs de la classe InetAddress

public byte [] getAddress ()

```
public class MyAddress {  
    public static void main (String args [ ]){  
try {  
    InetAddress ordlocal = InetAddress.getLocalHost ( );  
    byte [ ] adresse = ordlocal.getAddress ( ) ;  
    System.out.println ("Mon adresse est:" );  
    for (int i = 0; i < adresse.length; i++)  
    {        int octetNonSigne = adresse [i] < 0 ? adresse [i] + 256 : adresse[i] ;  
        System.out.print (octetNonSigne + "." );  
    }  
}  
catch (UnknownHostException e) {  
    System.out.println ("Désolé. Je ne connais pas mon adresse!" );}  
} // fin de la méthode main  
}  
} // fin de la classe
```



Acquérir des données grâce aux URL

Un **URL** (**U**niform **R**esource **L**ocator) est une *chaîne de caractère* qui désigne une ressource précise accessible par Internet ou Intranet.

Un URL est donc une *référence à un objet* dont le format dépend du protocole utilisé pour accéder à la ressource.

Plutôt que de stocker cet identificateur dans une simple chaîne, il est dès lors judicieux de le convertir en un objet dont les champs (protocole, adresse littérale, port, chemin d'accès, nom du fichier, section) sont individuellement manipulables.

La classe **java.net.URL** permet de gérer les URLs.

RETENEZ BIEN:

Les attributs de cette classe ne sont accessibles qu'aux autres membres du paquet *java.net*. Autrement dit, les classes *externes* ne peuvent accéder directement aux champs d'une instance URL. Acquérir ces valeurs nécessite donc l'initialisation des champs correspondants grâce aux constructeurs d'URL, puis d'en extraire les valeurs par le biais de méthodes appropriées (**getHost ()**, **getPort ()**,...).

Acquérir des données grâce aux URL

(Les différents constructeurs d' URL)

```
public URL (String url) throws MalformedURLException
```

C'est le plus simple des quatre constructeurs de la classe URL qui prend en paramètre une chaîne contenant l' URL.

Tous les constructeurs de la classe URL peuvent lever une **MalformedURLException**.

```
public class URLTestConstructeur {  
    public static void main (String args [ ]) {  
        try {  
            URL urlweb = new URL ("www.ucad.sn/ products.html");  
            System.out.println (urlweb);  
        }  
        catch (MalformedURLException e) { }  
    }  
}
```

Joseph NDONG UCAD DAKAR
SENEGAL

Acquérir des données grâce aux URL

public URL (String protocole, String hote, String fichier)

throws **MalformedURLException**

Ce constructeur prend en charge le protocole utilisé, le nom de l'hôte et le fichier à rapatrier. La valeur -1 est assigné au port, ce qui fait que c'est le port par défaut du protocole qui est utilisé.

N'omettez pas de saisir une barre oblique au début de la chaîne spécifiant le fichier, d'inclure le chemin, le nom de ce dernier et optionnellement une ancre nominale. **L'omission de la barre oblique est erreur très fréquente et difficilement décelable.**

```
try {  
    URL urlweb = new URL ("http", "www.ucad.sn" , "/ products.html#intro");  
    System.out.println (urlweb);  
}  
catch (MalformedURLException e) { }
```

Cette URL utilise le port par défaut de HTTP, 80.

Acquérir des données grâce aux URL



```
public URL (String protocole, String hote, int port, String fichier)  
    throws MalformedURLException
```

Dans les rares cas où le port par défaut du protocole utilisé ne convient pas, ce constructeur permet de spécifier explicitement le numéro du port adéquat sous la forme d'un entier.

```
try {  
    URL urlweb = new URL ("http", "www.ucad.sn" ,80, "/ products.html#intro");  
    System.out.println (urlweb);  
}  
catch (MalformedURLException e) { }
```

Acquérir des données grâce aux URL

public URL (URL u, String s) throws MalformedURLException

Ce constructeur, qui génère un URL *absolu* à partir de l'URL *relatif* fourni, est probablement celui auquel on fera le plus souvent appel.

Supposons que le document HTML <http://www.macfaq.com/index.html> révèle un lien vers un fichier de nom **vendor.html**, sans plus d'information: pour compléter ces renseignements, on confiera au constructeur l'URL du document contenant le lien afin qu'il produise le nouvel URL <http://www.macfaq.com/vendor.html>.

```
try {  
    URL u1 = new URL ("http://www.macfaq.com/index.html");  
    URL u2 = new URL (u1, "vendor.html");  
  
    System.out.println (urlweb);  
}  
catch (MalformedURLException e) { }
```

Ce constructeur permet de traiter successivement dans une boucle une liste de fichiers résidant dans le même répertoire.



```
public URL (URL u, String s) throws MalformedURLException
```

Ce constructeur peut également servir à générer un URL *relatif* pointant vers le répertoire d'origine de l'applet, dont on extrait le nom grâce à la méthode `getDocumentBase ()` ou la méthode `getCodeBase ()`. Voici un exemple:

```
import java.net.*;
import java.applet.Applet ;
public class URLrelatifApplet {
    public void init ( ) {
        URL u1, u2;
        u1 = getDocumentBase ( ) ;
        try {
            u2 = new URL (u1, "produits.html"); // nouveau URL relatif généré
        }
        catch (MalformedURLException e) {
            Sytem.err.println (e.getMessage ( ));
        }
    }
}
```


Comment fractionner un URL

La classe URL contient six (6) *champs*: **protocol**, **host**, **port**, **file**, **URLStreamHandler** et **l'ancre nominale** (nommée souvent « section » ou « ref »).

Tous ces champs sont privés exceptés *URLStreamHandler* qui reste accessible aux autres membres du paquet *java.net*.

Les cinq méthodes suivantes en permettent l'accès en mode « lecture seule ».

```
getProtocol ( ) // renvoie sous forme de chaîne la portion de l'url décrivant le protocole
getHost ( )    //renvoie une chaîne contenant le nom d'hôte de l'url,
getPort ( )    // renvoie un entier correspondant au numéro de port spécifié dans l'url
getFile ( )    // renvoie une chaîne contenant le nom et le chemin du fichier de l'url
               // si aucun fichier ne figure dans l'url, alors elle renvoie une barre oblique
getRef ( )     //renvoie l'ancre nominale mentionnée dans l'url
               // si aucune ancre n'est présente, elle renvoie « null ».
```

Exemple de fractionnement d'un URL

```
public class FractURL {  
    public static void main(String[] args) {  
  
        try {  
            URL u = new URL ("http://www.ncsa.uiuc.edu /demoweb/produits.html#A1.3.3.3");  
            System.out .println("URL "+u);  
            System.out .println("protocole "+u.getProtocol () );  
            System.out .println("nom d hôte "+u.getHost() );  
            System.out .println("numéro de port "+u.getPort() );  
            System.out .println("chemin d' acces et nom du fichier "+u.getFile() );  
            System.out .println("ancree "+u.getRef() );  
        }  
        catch (MalformedURLException r){  
            System.err .println(" impossible de traiter cet URL");  
        }  
    }  
}
```



Extraction de données d'un URL

Le véritable « jeu » dans la manipulation des url est dans l'extraction des informations contenues dans le fichier correspondant.

Les trois méthodes issues de **java.net.URL** qui permettent d'accéder à ces informations sont les suivantes:

```
public final InputStream openStream ( ) throws IOException  
public URLConnection openConnection ( ) throws IOException  
public final Object getContent ( ) throws IOException
```



Extraction de données d'un URL

```
public final InputStream openStream ( ) throws IOException
```

Cette méthode, après s'être connectée à la ressource référencée dans l'URL, **prend en charge la procédure d'authentification entre le client et le serveur** puis **ouvre un `InputStream`**. Ce dernier extrait les données brutes (non interprétées) contenues dans le fichier désigné par l'URL: données ASCII d'un fichier texte, code HTML d'un document HTML, données graphiques binaires d'une image, etc.

Les informations liées au protocole (en-têtes HTTP, par ex.) n'y figurent pas et visualiser de telles informations nécessite l'usage de la méthode *openConnection* que nous étudierons dans la suite de ce cours.

Exemple d'extractions avec **openStream()**

```
public class AfficheSource {
    public static void main(String[] args) {
        String lignecourante;
        URL u;
        try { u = new URL ("http://127.0.0.1:8080/essai.txt") ;
            // convertir l url en un DataInputStream
            try { InputStream is = u.openStream ( ) ;
                DataInputStream donneesText = new DataInputStream(is);

                try { while ((lignecourante = donneesText.readUTF ( ) ) != null ) {
                    System.out.println (lignecourante); }
                    } catch (IOException e) { System.err.print ("Erreur lecture") ;}
                } catch (IOException r) { System.err .println("Erreur liaison flux ") ;
                    r.printStackTrace() ;}
            } catch (MalformedURLException g){
                System.err .println ("impossible de traiter l'url") ;}
        }
    }
```

Extraction de données d'un URL

```
public URLConnection openConnection ( ) throws IOException
```

Cette méthode ouvre un socket vers l'url spécifié et renvoie un objet **URLConnection** qui représente une connexion ouverte vers une ressource réseau. Elle renvoie une **IOException** si l'appel échoue.

Cette méthode permet donc une communication directe avec le serveur.

Tout ce que transmet le serveur est dès lors accessible:
le document lui-même dans sa forme brute (HTML, texte, données binaires),
mais aussi les en-têtes requis par le protocole mis en œuvre.

```
try { URL u = new URL ("http://127.0.0.1:8080/essai.txt") ;  
    try { URLConnection uc = u.openConnection ( ) ;  
        } catch (IOException e) { ;}  
  
    } catch (MalformedURLException r) {  
        r.printStackTrace() ;}
```

Extraction de données d'un URL

```
public final Object getContent ( ) throws IOException
```

Cette méthode offre un autre moyen de télécharger les informations référencées par un URL, en retournant dans un objet approprié: **InputStream** dans le cas d'un *fichier HTML ou ASCII*, **ImageProducer** dans le cas d'une *image GIF ou JPEG*.

getContent () détermine le *type d'objet* à instancier grâce à l'analyse du champ Content-type de l'en-tête MIME émanant du serveur. Si ce dernier ne transmet pas d'en-tête MIME ou que le format spécifié n'est pas reconnu, **getContent ()** renvoie une **ClassNotFoundException**, ou encore une **IOException** lorsqu'il lui est impossible d'extraire l'objet.

L'opérateur **instanceof** permet de vérifier si l'objet est de type **InputStream** ou **ImageProducer** et de distinguer ainsi entre fichiers textes et images.



La classe `java.net.URLEncoder`

L'encodage est un processus qui permet d'avoir un programme portable vu la disparité des systèmes d'exploitation qui représentent certains caractères différemment.

La procédure d'encodage, consiste à remplacer chaque caractère autre que majuscule ou minuscule, chiffre ou caractère de soulignement par le code ASCII correspondant, représenté par deux chiffres hexadécimaux précédés du signe pourcent (%).


La classe `URLEncoder` se compose d'une seule méthode:
`public static String encode (String s)` qui assure l'encodage de la chaîne fournie.
La sortie de cette méthode contient la chaîne dûment encodée.

Par exemple:

```
System.out.println (URLEncoder.encode ("voici quelques espaces");
```

donnera: **voici+quelques+espaces**

La classe **Socket**: les constructeurs



```
public Socket (String hote, int port) throws UnknownHostException,  
                                           IOException
```

Ce constructeur génère un socket TCP sur le port indiqué de l'hôte visé, puis entreprend de s'y connecter.

```
try {  
    Socket versUca = new Socket ("www.ucad.sn", 80);  
}  
catch (UnknownHostException e) {  
    e.printStackTrace();  
}  
catch (IOException e) {  
    System.err.println (e.getMessage ());  
}
```

Les causes liées à l'échec d'une connexion sont diverses et variées: machine cible refusant toute connexion, problème lié à la connexion Internet, erreur de routage des paquets, etc.

La classe **Socket**: les constructeurs



```
public Socket (InetAddress hote, int port) throws IOException
```

Ce constructeur génère un socket TCP sur le port spécifié de l'hôte visé et tente de s'y connecter. Le nom de l'hôte est fourni par un objet **InetAddress**.

Ce constructeur gère seulement l'exception **IOException** puisque l'autre c à d le **UnknownHostException** est pris en compte dès la création de l'objet **InetAddress**.

```
InetAddress uca = null; Socket soc;  
try {  
    uca = InetAddress.getByName ("www.ucad.sn");  
}  
catch (UnknownHostException e) { e.printStackTrace();}  
try {  
    soc= new Socket (uca, 80);  
}  
catch (IOException e) { System.err.println (e.getMessage ( )); }
```

La classe **Socket**: les constructeurs



```
public Socket (String hote, int port, InetAddress interface,  
int portLocal) throws IOException
```

Avec ce constructeur, le numéro du port et le nom de l'hôte sont fournis dans les deux premiers paramètres et la connexion s'établit à partir de l'interface réseau physique (une carte Ethernet, par ex.) ou virtuelle (Système multi adresse) et du port local indiqué dans les deux derniers paramètres.

Si le portLocal vaut 0, la machine choisit aléatoirement un port disponible (parfois appelé anonyme) compris entre 1024 et 65535.

```
InetAddress sunsite ; Socket soc;  
try {      sunsite = InetAddress.getByName ("sunsite.unc.edu"); }  
catch (UnknownHostException e)  { e.printStackTrace();}  
  
try {      soc = new Socket ("www.oreilly.com", 80, sunsite,0);    }  
catch (IOException e) { System.err.println (e.getMessage ( )); }
```

Là on exécute un programme sur www.oreilly.com et désirons nous assurer que notre connexion passe par l'interface FDDI 100 Mb/s (**sunsite.unc.edu**) au lieu de l'interface Ethernet 10 Mb/s.

La classe **Socket**: les constructeurs



```
public Socket (InetAddress hote, int port, InetAddress interface,  
int portLocal) throws IOException
```

Ce constructeur est identique au précédent sauf qu'ici le nom de l'hôte est un objet `InetAddress`.

```
InetAddress sunsite, ora ; Socket soc;  
try {  
    sunsite = InetAddress.getByName ("sunsite.unc.edu");  
    ora      = InetAddress.getByName (" www.oreilly.com ");  
}  
catch (UnknownHostException e)  { e.printStackTrace();}  
  
try {  
    soc = new Socket (ora, 80, sunsite,0);  
}  
  
catch (IOException e) { System.err.println (e.getMessage ( )); }
```



La classe **Socket**: les constructeurs

La classe socket offre deux autres constructeurs protégés qui initialisent la classe mère Sans connecter le socket. On y recourt lors de la création de classes dérivées, en vue d'implanter par exemple un Socket original capable de chiffrer des transactions ou d'interagir avec le serveur partagé local.

protected Socket ()

protected Socket (SocketImpl impl)

Se renseigner sur un **Socket**: les méthodes

Un socket possède un seul champ qui contient un *SocketImpl*. Les autres champs visibles appartiennent à cet objet. Pour obtenir des renseignements sur un socket plusieurs méthode sont disponibles:

*/*renvoie le nom de l'hôte auquel le socket spécifié est connecté ou, si la connexion est close, le nom de l'hôte visé*/*

public InetAddress getAddress ()

*/*indique le numero du port de la machine distante auquel un socket est (ou sera) connecte*/*

public int getPort ()

/ indique le numero du port de la machine local auquel un socket est connecte */*

public int getLocalPort ()

*/*indique l'interface reseau à laquelle un socket donne est rattache*/*

public InetAddress getAddress ()

Se renseigner sur un **Socket**: les méthodes

public InputStream **getInputStream () throws IOException**

Cette méthode renvoie un flux d'entrée brutes grâce auquel un programme peut lire des informations à partir d'un socket. Il est d'usage de lier cet *InputStream* à un autre flux offrant davantage de fonctionnalités (un *DataInputStream*, par ex.)

```
Socket s;  
String hote = "localhost" ;  
DataInputStream flux;  
try {  
    s = new Socket (hote, 80);  
    flux = new DataInputStream(s.getInputStream( ));  
}  
catch (UnknownHostException e) {  
    System.err.println(e);  
}  
catch (IOException e) { }
```

Se renseigner sur un **Socket**: les méthodes

public OutputStream **getOutputStream () throws IOException**

Cette méthode instancie un *OutputStream* brut permettant à une application d'écrire des données à l'autre extrémité du socket. Il est conseillé d'associer ce flux à une classe plus puissante comme *DataOutputStream* avant d'en user.

Fermer un socket

La méthode chargée de déconnecter un socket se nomme **close ()**; elle s'emploie comme suit:

leSocket.close () ;

Même si le socket est fermé, l'*InetAddress* de l'objet socket, le *numéro du port*, l'*adresse locale* et le *numéro du port local* demeurent accessibles via les méthodes d'accès prévues. Par contre, le fait de vouloir lire l'*InputStream* ou d'écrire dans l'*OutputStream* en appelant *getInputStream ()* ou *getOutputStream ()* déclenche une *IOException*.



Socket Server

Les passages précédents montraient, comment, grâce aux sockets, un client peut se connecter à un serveur à l'écoute des requêtes de connexion. Nous allons compléter ce cours en nous intéressant à un autre type de socket (socket serveur) basé sur la classe **ServerSocket**. Un socket client transmet une requête tandis qu'un socket serveur attend une requête.



La classe **ServerSocket**

La classe ***ServerSocket*** est utilisée côté serveur: elle renferme tout le nécessaire pour la réalisation de programmes serveur. Elle attend simplement les appels du (es) client (s). C'est un objet de type Socket qui prend en charge la transmission des données.

Ses constructeurs génèrent chaque nouvel objet *ServerSocket*, ses méthodes interceptent les demandes de connexion sur les ports spécifiés, retournent un socket une fois celle-ci établie afin de permettre les échanges de données, ou assurent des opérations plus communes (conversion de format...).



La classe **ServerSocket**

Comment un serveur gère une connexion et s'y prépare?

1. Le constructeur *ServerSocket* () instancie un nouvel objet *ServerSocket* affecté à un port donné, *port sur lequel le serveur écoute et reçoit les requêtes du (des) client (s)*.
2. Grâce à la méthode **accept ()**, l'objet guette les requêtes de connexion adressées à ce port. Cette méthode intervient seulement lorsqu'un client tente d'entrer en communication avec le serveur, auquel cas elle renvoie un objet *Socket* pour relier les deux machines.
3. Les méthodes **getInputStream ()** et/ ou **getOutputStream ()** du socket délivrent, selon le type du serveur, les flux d'entrée et de sortie indispensables aux échanges.
4. Le serveur et le client interagissent selon le protocole choisi pendant le temps de la connexion.
5. Le serveur ou le client clos la connexion
6. Le serveur reprend sa position d'attente (retour à l'étape 2).

Les clients reçoivent les données provenant du serveur sur un port libre automatiquement choisi par le système.



La classe ServerSocket: les constructeurs

*/*instancie un socket server et l'affecte au port indiqué, si le port = 0, alors il est sélectionné par le système*/*

public ServerSocket (int port) throws IOException, BindException

*/*met en place un socket server sur le port indiqué et attribue à la file d'attente le nombre de requêtes en attente à partir duquel toute nouvelle demande est refusée*/*

public ServerSocket (int port, int taillefile) throws IOException, BindException

*/*instancie un socket server et l'affecte au port indiqué. CE constructeur est associé à une adresse IP locale et gère uniquement les requêtes destinées à cette adresse particulière*/*

public ServerSocket (int port, int taillefile, InetAddress adrAttache) throws IOException

*/*ce constructeur protégé est réservé aux classes dérivées de ServerSocket qui souhaitent disposer d'un SocketImpl spécial pour interagir avec un serveur partagé ou implémenter des protocoles de sécurité*/*

protected ServerSocket ()



Accepter et clore une connexion: les méthodes **accept ()** et **close ()**

public Socket **accept ()** **throws** IOException.

Cette méthode ne doit être appelée qu'une fois les préparatifs terminés. *Elle fige l'exécution du programme serveur dans l'attente qu'un client se connecte et*, le cas échéant, renvoie un objet Socket, dont les méthodes **getInputStream ()** et **getOutputStream ()** permettent de communiquer avec le requérant.

```
try { ServerSocket leserveur = new ServerSocket ( 5778);  
    while ( true) {  
        Socket conCourante = leserveur.accept();  
        PrintStream versclient = new PrintStream (  
                                                    conCourante.getOutputStream ( ));  
        versclient.println ("merci de votre visite ");  
        conCourante.close ( );  
    }  
}  
catch (IOException e){ leserveur.close ( ) ; System.err.println (e);}
```



Accepter et clore une connexion: les méthodes **accept ()** et **close ()**

public void **close ()** **throws** IOException.

En fermant le socket serveur, cette méthode libère le port, qui peut dès lors être utilisé par tout autre programme du système.

La fermeture d'un objet **ServerSocket** n'a pas les mêmes conséquences que celle d'un **Socket**: la première libère le port correspondant sur le système local, tandis que la seconde rompt la connexion entre les deux machines.

Les sockets serveur sont automatiquement clos à l'arrêt du programme. Mais il faut avoir l'habitude de les fermer explicitement.



Quelques méthodes **get**

public InetAddress **getInetAddress ():**

Cette méthode renvoie l'adresse associée au serveur (sur le système local), soit la sortie de **InetAddress.getLocalHost ()** si l'hôte possède une seule adresse IP ou, plus rarement, une de ses adresses s'il en possède plusieurs.

Cette méthode est très utile si on veut lancer un client et un serveur sur la même machine locale; on écrira:

```
Socket serv = new Socket (InetAddress.getLocalHost ( ), 4444);
```

public int **getLocalPort ():**

Lorsque le numéro de port vaut 0, les constructeurs *ServerSocket* mettent en place un socket serveur sur un port choisi par le système. Cette méthode permet d'en connaître le numéro.

Communication client-serveur TCP

serveur

ServerSocket

serv = new ServerSocket (**5600**)

J'ouvre le port 5600 et
J'attends les requêtes
entrantes

Socket versclient = serv.**accept** ()

InputStream is = versclient.getInputStream ();
OutputStream os = versclient.getOutputStream ();

- généralement tamponner is et os
- réceptionner des données
- envoyer des réponses
- clore la socket vers le client

client

Socket versServeur =
new Socket ("192.168.0.12", **5600**)

InputStream is = versServeur.getInputStream ();
OutputStream os = versServeur.getOutputStream ();

- généralement tamponner is et os
- envoyer des données
- recevoir des réponses
- clore la socket vers le serveur

Débloque **accept** ()

Exemple de serveur simple TCP

```
package autre;
public class ServeurTCP {
    ServerSocket server = null;
    Socket client = null;
    PrintWriter out = null;
    BufferedReader in = null;
    String line;
    ServeurTCP () {
        try {
            server = new ServerSocket (4444);
        }
        catch (IOException e) {
            System.out.println ("Could not listen on port 4444");
            System.exit (-1);
        }
    }
}
```

Exemple de serveur TCP

```
try {System.out.println ("Serveur en attente sur le port: "+server.getLocalPort ( ));
    client = server.accept ( );
    System.out.println ("Traitement avec le client "+client.getRemoteSocketAddress());
}
catch (IOException e) {
    System.out.println("Accept failed: 4444");
    System.exit(-1);
}
try {
    in = new BufferedReader (new InputStreamReader (client.getInputStream ( )));
    out = new PrintWriter (client.getOutputStream ( ), true);
}
catch (IOException e) {
    System.out.println ("flux errone");
    System.exit(-1);
}
```

Exemple de serveur TCP

```
try { String mesrecu = in.readLine ( );
    if (mesrecu != null) out.println ("Merci de votre visite, j' ai bien reçu:"+mesrecu) ;
    }
catch (IOException e) {
    System.out.println ("Read failed");
    System.exit (-1);
    }
}

try { in.close ( );
    out.close ( );
    server.close ( );
    }
catch (IOException e) { System.out.println ("Could not close."); System.exit(-1);
    }
}

public static void main (String a [ ]){
    new ServeurTCP ( );
    }
} // fin de la classe server
```



Exemple de client TCP

```
public class SocketClient {
    Socket socket = null;
    PrintWriter out = null;
    BufferedReader in = null;
    String line;
    SocketClient () {
        try {
            socket = new Socket (InetAddress.getLocalHost( ), 4444);
            out = new PrintWriter (socket.getOutputStream ( ), true);
            in = new BufferedReader( new InputStreamReader (socket.getInputStream ( )));
        }
        catch (UnknownHostException e) {
            System.out.print (" Unknown host:");
        }
        catch (IOException e) {
            System.out.print (" No I/O");
        }
    }
}
```

Exemple de client TCP

```
/*envoyer vers le serveur*/
public void envoyer (String mes){

    out.println (mes) ;
    System.out .println ("Le client a envoye : "+mes) ;

}
/*ce que j' ai reçu du serveur*/
public void reception ( ){
    try {
        String rec = in.readLine ( );
        System.out .println("Depuis le serveur: "+rec) ;
    }
    catch (IOException e){ }
}
} // fin de la classe client
```



Tester le client/serveur

```
public class TestClientServeur {  
    public static void main (String [ ] args) {  
  
        SocketClient c = new SocketClient ( );  
        c.envoyer ("bonjour") ;  
        c.reception ( );  
    }  
}
```

Sortie de l'exemple:

Le client a envoye : **bonjour**

Depuis le serveur: Merci de votre visite, j' ai bien reçu: **bonjour**



Transmission d'un objet sérialisé

Envoi d'un objet sérialisé

La sérialisation permet de sauvegarder l'état interne d'un objet, c'est-à-dire ses variables, puis de les récupérer dans un flux. On peut donc envoyer les informations de cette instance dans le réseau après avoir utilisé la sérialisation.

Pour ce faire:

On instancie un socket en spécifiant par exemple le port et l'adresse IP et en acceptant une connexion entrante.

On déclare un *BufferedOutputStream* qui prend le flux du socket en paramètre (*socket.getOutputStream()*).

La sérialisation d'un objet est effectuée lors de l'appel de la méthode *writeObject* sur un objet implémentant l'interface *ObjectOutput* (par exemple un objet de la classe *ObjectOutputStream*) en passant en paramètre à cette méthode l'objet que l'on veut sérialiser, et en paramètre au constructeur *ObjectOutputStream* le flux sur lequel on veut envoyer l'objet sérialisé.



Réception d'un objet sérialisé

La réception de l'objet sérialisé fonctionne quasiment comme l'envoi.

On crée un socket et on déclare un *BufferedInputStream* qui prend en paramètre le flux entrant du socket .

La désérialisation est effectuée en appelant la méthode *readObject* sur un objet implémentant l'interface **ObjectInput** (par exemple un objet de la classe *ObjectInputStream*).



Exemple de sérialisation réseau

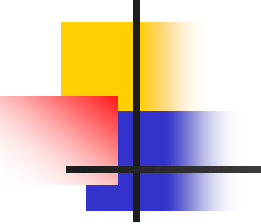
L'exemple comprend trois classes: une classe Personne dont une instance sera sérialisée et envoyée dans le réseau. La classe Envoi se chargera d'envoyer cette instance de personne. La classe Reception recevra l'instance et affichera ses différents champs.



```
package serveururtcp;
import java.net.*;
import java.io.*;
public class Envoi implements java.io.Serializable {
    Socket sock;

    BufferedOutputStream tampon;
    ObjectOutput ecrit;
    public Envoi (Socket sock) {
        try {
            this.sock = sock;
            tampon = new BufferedOutputStream( sock.getOutputStream ( ));
            ecrit = new ObjectOutputStream(tampon);
        }
        catch (Exception er){ }
    }
    public void envoyer (Personne p){
        try {
            ecrit.writeObject (p);
            ecrit.flush ( );

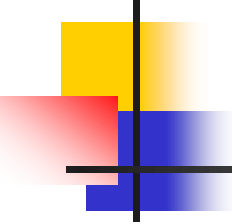
        }
        catch (Exception e){}
    }
}
```



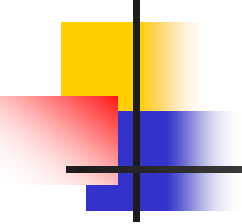
```

public class Reception implements java.io.Serializable {
    Socket sock;
    ServerSocket s;
    BufferedInputStream tampon;
    ObjectInput lu;
    public Reception (ServerSocket s) {
        try{
            this.s = s;
            sock = this.s.accept ( ) ;//bloquant
            tampon = new BufferedInputStream (sock.getInputStream ( ));
            lu = new ObjectInputStream (tampon);
        }
        catch (Exception er){}
    }
    public File recevoir ( ){
        try{
            Personne p = (Personne) lu.readObject() ;
            return new Serializer ( ).serializerUnePersonne(p);
            ecrit.close ( ) ;
        }
        catch ( Exception e){ return null;}
    }
}

```



```
public class TestEnvoi {  
    public static void main (String [ ] args) {  
        try {  
            Personne p = new Personne("Joe", "NDONG", 123);  
  
            Envoi e = new Envoi ( new Socket("localhost", 678));  
            e.envoyer (p);  
        }  
        catch (Exception d ) { }  
    }  
}
```



```
public class TestReception {  
    public static void main (String[] args) {  
        try {  
            Reception r = new Reception (new java.net.ServerSocket (678));  
            File f = r.recevoir ( );  
            FileInputStream fic = new FileInputStream (f);  
            ObjectInputStream ois = new ObjectInputStream (fic);  
            Personne p = (Personne) ois.readObject ( );  
            System.out .println (p .getPrenom ( ) ) ;  
            System.out .println (p.getNom ( ) ) ;  
            System.out .println (new Integer (p.getTaille ( ) ) ) ;  
  
        }  
        catch (Exception e){}  
    }  
}
```



Sockets et Datagrammes UDP

Les sections précédentes faisaient allusion au protocole TCP, conçu pour une transmission fiable des données. Mais cette fiabilité coûte cher puisque la clôture des connexions TCP exige le plus souvent un certain temps, surtout pour un protocole comme HTTP, qui requiert en général des transferts nombreux et courts.

UDP (**U**ser **D**atagram **P**rotocol) est un protocole qui offre un transfert très rapide au détriment de la fiabilité. Rien ne permet de dire que les données transmises arriveront au destinataire dans l'ordre initial, ni même si elles lui parviendront.

Un client UDP qui envoie une courte requête UDP à un serveur la considérera perdue si aucune réponse ne lui parvient dans la durée impartie (par ex. le DNS applique cette technique).

Le mode UDP est connecté moins fiable mais permet de gagner en performances. Il est très utilisé dans le monde multimédia, permettant une transmission rapide de données de même nature et en paquets.



Sockets et Datagrammes UDP

Java implante UDP à travers deux classes: **DatagramPacket** et **DatagramSocket**. La première rassemble dans les paquets UDP appelés **datagrammes** des octets de données en partance, et extrait le contenu de datagrammes reçus. Les objets *DatagramSocket*, quant à eux, envoient et réceptionnent les datagrammes UDP.

En clair, *on expédie des données* en les plaçant dans un *DatagramPacket* puis *on les envoie* grâce à un *DatagramSocket*, et *on reçoit des données* dans un objet *DatagramPacket* transmises par un *DatagramSocket*.



Sockets et Datagrammes UDP

La gestion des sockets est très simple: les datagrammes UDP incluent tous les détails, y compris l'adresse de destination. **Il suffit d'indiquer donc au socket le numéro du port par lequel les informations transitent.**

Cette technique est à l'opposé de celle qu'appliquent les classe *Socket* et *ServerSocket*.

UDP n'a aucune notion de ce qu'est un *socket server*. **Un même type de socket sert à la fois à l'envoi et la réception de données.**

Avec UDP, les paquets sont traités *individuellement*: la perte d'un paquet entraîne la disparition de l'intégralité des données qu'il contient.

La classe **DatagramPacket**: les constructeurs

Cette classe dispose de deux **constructeurs**. Le premier s'emploie pour le rapatriement de données depuis l'Internet. Le second est destiné à l'envoi d'informations.

Ces deux constructeurs renvoient des objets aux usages différents. Pour le premier constructeur, les paramètres passés en arguments sont **le nom et la longueur du tableau d'octets voué au stockage des données**. Seuls ces deux paramètres sont passés avant réception d'informations. *Le tampon de données doit être alors vide.*

Le second constructeur génère un datagramme avant envoi. Aux paramètres précités, il convient d'ajouter **l' InetAddress et le port de destination** du paquet. Ici *le tampon n'est pas vide*, il contient les données à expédier.

La classe **DatagramPacket**: les constructeurs

```
public DatagramPacket (byte tampon [ ], int longueur)
```

Ce constructeur instancie un nouveau *DatagramPacket* à recevoir. **tampon** est un tableau d'octets destiné à *recueillir* les informations.

Si un paquet arrive, le premier octet qu'il contient est placé dans **tampon [0]**, le second dans **tampon [1]** et ainsi de suite.

Ce constructeur provoque une **IllegalArgumentException** si **longueur > tampon.length** .

La longueur maximale d'un datagramme ne peut dépasser 64 Ko (65535 octets).

Huit (8) octets y sont alloués à l'en-tête UDP et au moins vingt (et parfois jusqu'à 60) à l'en-tête IP. Par conséquent, un datagramme ne peut renfermer plus de **65 507 octets de données**.

La classe **DatagramPacket**: les constructeurs

```
public DatagramPacket (byte tampon [ ], int longueur  
                        InetAddress adr, int port)
```

Ce constructeur produit un nouveau *DatagramPacket* destiné à un hôte distant. Le paramètre tampon indique les données à transmettre et le paramètre longueur le nombre d'octets à expédier. Si **longueur > tampon.length**, ce constructeur lance une **IllegalArgumentException**.

L'objet *InetAddress* adr pointe vers la machine distante et l'entier port désigne le numéro du port de destination.

REMARQUE:

Il est d'usage de convertir les données en tableau d'octets avant de placer celui-ci dans un tampon avant d'instancier un objet *DatagramPacket*.

Cette conversion peut utiliser la méthode **getBytes ()** de **java.lang.String** s'il s'agit d'un texte ASCII. (*cf. l'exemple suivant*).



```
public DatagramPacket (byte tampon [ ], int longueur  
                        InetAddress adr, int port)
```

```
public class ExpediturUDP {  
    public static void main (String args[ ]) {  
        String s = "message a envoyer";  
        byte [ ] donnees = new byte [s.length ( )];  
        s.getBytes ( ); //conversion des données en octets  
        try {  
            InetAddress adr = InetAddress.getByName ("www.ucad.sn");  
            int port = 7;  
            DatagramPacket dp = new DatagramPacket (donnees, donnees.length, adr, port);  
        }  
        catch (UnknownHostException e) { }  
    }  
}
```



Les méthodes **get**

Les quatre méthodes ci après permettent d'extraire plusieurs composants d'un datagramme (charge utile, champs de l'en-tête).

Un datagramme se compose tout d'abord d'un en-tête IP indiquant l'adresse de la machine expéditrice et celle de l'hôte destinataire. Vient ensuite un en-tête UDP codé sur 8 octets.

En vertu du RFC768, les deux premiers octets contiennent un entier désignant le numéro du port d'expédition, les deux suivants stockent le numéro du port de destination, et les troisième et quatrième paires d'octets contiennent l'une, la longueur du datagramme, le présent en-tête compris, et l'autre une somme de contrôles destinée à garantir un transfert fiable, quoique souvent remplacée par la valeur 0.

Tout ce qui suit ces deux en-têtes représente la charge utile du datagramme.



Les méthodes **get**

```
public InetAddress getAddress ( )
```

Cette méthode renvoie un objet `InetAddress` caractérisant l'adresse de *l'hôte distant ou local*, selon qu'il s'agit d'un datagramme reçu par la machine locale ou bien créé par elle. On l'emploie surtout pour connaître l'adresse de l'expéditeur d'un datagramme UDP avant d'y répondre.

```
public int getPort ( )
```

L'entier que renvoie cette méthode représente le numéro du port de provenance ou de destination du datagramme, selon qu'il s'agit d'un datagramme reçu par la machine locale ou bien créé par elle en vue d'être expédié.

Les méthodes **get**

```
public byte [ ] getData ( )
```

Cette méthode produit un tableau contenant les octets de données dont se compose le datagramme. Ces informations doivent en principe être converties pour que le programme puisse les manipuler. Pour transformer ce tableau d'octets, on utilise le constructeur `String (...)` suivant:

```
public String (byte [ ] tampon, int octet_sup, int debut, int nb_octets):
```

tampon → tableau d'octets contenant les données

octet_sup → ce constructeur considère que chaque octet représente un caractère ASCII, mais vu que les chaînes Java se composent de caractères Unicode longs de deux octets, il est donc nécessaire d'ajouter un octet supplémentaire à chaque caractère, c'est le rôle du paramètre *octet_sup* qui vaut presque toujours 0.

debut → premier élément du tableau à convertir, il équivaut toujours à 0.

nb_octets → nombre d'octets à traiter qui n'est pas nécessairement la taille du tableau. La méthode `getLength ()` permet de connaître le nombre exact d'octets stockés dans le tampon.

Voici un exemple:

```
String s = new String (dp.getData ( ),0,0, dp.getLength ( ));
```



```
public byte [ ] getData ( )
```

La classe **java.util.StringTokenizer** vous aidera à analyser les chaînes arborant un format ou une spécification spécifique.

La conversion de datagrammes non ASCII en données Java est plus difficile; une approche possible consiste à convertir le tableau d'octets par **getData ()** en objet **ByteArrayInputStream** grâce au constructeur:

```
public ByteArrayInputStream (byte [ ] tampon, int decalage, int nb_octets)
```

Il est important de délimiter *la portion du tampon concernée* grâce aux paramètres **decalage** et **nb_octets**.

Lors de la conversion d'un datagramme en objet *InputStream*, **decalage** vaut toujours 0 et **nb_octets** s'obtient au moyen de la méthode **getlength ()** du *DatagramPacket* :

ByteArrayInputStream bis = **new**

```
ByteArrayInputStream (dp.getData ( ),0,dp.getLength ( ));
```

Le *ByteArrayInputStream* peut ensuite être couplé à un **DataInputStream**.

Les méthode **readInt ()**, **readLong ()** et **readChat ()** permettent alors d'accéder aux données.



Les méthodes **get**

```
public byte [ ] getLength ( )
```

Cette méthode donne le nombre d'octets de données contenues dans le datagramme. Si ce dernier provient du réseau, l'entier retourné est en principe égal à la longueur du tableau indiqué par **getData ()** (précisément **getData().length**), mais parfois inférieur à celle-ci si le datagramme est l'œuvre du système local.



La classe **DatagramSocket**

La mise en place d'un *DatagramSocket* permet d'expédier ou de recevoir un objet issu de la classe *DatagramPacket*.

La classe *DatagramSocket* dispose de deux constructeurs aux usages spécifiques.

Chaque *DatagramSocket* est attaché à un port local sur lequel il guette l'arrivée de données et dont le numéro est consigné dans l'en-tête des datagrammes en partance. Ce renseignement importe peu pendant la rédaction du client, d'où l'appel d'un Constructeur laissant le système assigner un port disponible (dit « anonyme »).

La réponse du serveur transite par le port local d'origine du datagramme.

Lorsque vous rédigez un serveur, les clients ont besoin de connaître le numéro du port sur lequel le serveur épie l'arrivée des datagrammes; par conséquent le serveur doit mentionner cette information lorsqu'il crée un *DatagramSocket*.

A ce détail près, sockets client et sockets serveur sont identiques et ne diffèrent que lorsque le port mis en œuvre est anonyme ou prédéfini.

La classe `DatagramSocket`: les constructeurs

```
public DatagramSocket ( ) throws SocketException
```

Ce constructeur instancie un socket rattaché à un port anonyme (le port de destination fait partie du *DatagramPacket* et non du *DatagramSocket*).

Cette méthode est utilisée lorsqu'un client entame une conversation avec un serveur: le port n'a pas d'importance puisque le serveur expédie sa réponse vers le port d'origine du datagramme.

Si vous voulez connaître le numéro de ce port local, utilisez la méthode `getLocalPort ()`.

Un même socket peut s'employer pour réceptionner les datagrammes émis par un serveur.

L'échec de ce constructeur entraîne une `SocketException`.

*/*mise en place d'un DatagramSocket sur un port anonyme*/*

```
public class MonDatagramSocket {  
    public static void main (String args [ ]) {  
        try { DatagramSocket leclient = new DatagramSocket ( ); }  
        catch (SocketException e) { }  
    }  
}
```

La classe DatagramSocket: les constructeurs

```
public DatagramSocket (int port ) throws SocketException
```

Si les serveurs guettaient sur des ports anonymes, aucun client ne pourrait les contacter. Donc ce constructeur instancie un socket qui surveille l'arrivée de datagrammes sur le port passé en paramètre.

L'échec de l'opération déclenche une SocketException qui traduit tantôt l'indisponibilité du port visé, tantôt des droits d'accès insuffisants.

*/*recherche de ports UDP locaux*/*

```
public class RecherchePortsUDPLocaux {  
    DatagramSocket leServeur;  
    public static void main (String args [ ]) {  
        for (int i = 1024; i < 65535; i++) {  
            try { leServeur = new DatagramSocket (i ); leServeur.close ( );}  
            catch (SocketException e) { }  
            System.out.println ("Un serveur est rattaché au port " +i);  
        }  
    }  
}
```

La classe **DatagramSocket**: les constructeurs

```
public DatagramSocket (int port, InetAddress adr )  
    throws SocketException
```

Ce constructeur est surtout utilisé sur des machines multi adresses. Il crée un socket qui guette l'arrivée de datagrammes sur un port et via une interface réseau donnée.

Comme dans le cas des sockets TCP, UNIX interdit à quiconque de mettre en place un **DatagramSocket** sur un port inférieur à 1024, hormis au super utilisateur.

L'objet **InetAddress** représente l'une des adresses réseau de l'hôte.



Émission et réception des datagrammes

Emission: `public void send (DatagramPacket dp) throws IOException`

Une fois instanciés, le **DatagramPacket** et le **DatagramSocket**, le paquet peut être expédié grâce à la méthode **send ()** du socket. Ainsi on a:

```
monSocket.send (monPaquet);
```

L'envoi d'un paquet peut lever une *IOException* souvent du au fait que le **SecurityManager** n'autorise pas les échanges avec l'hôte visé. Aucune exception ne se pose si le paquet n'atteint pas la destination.

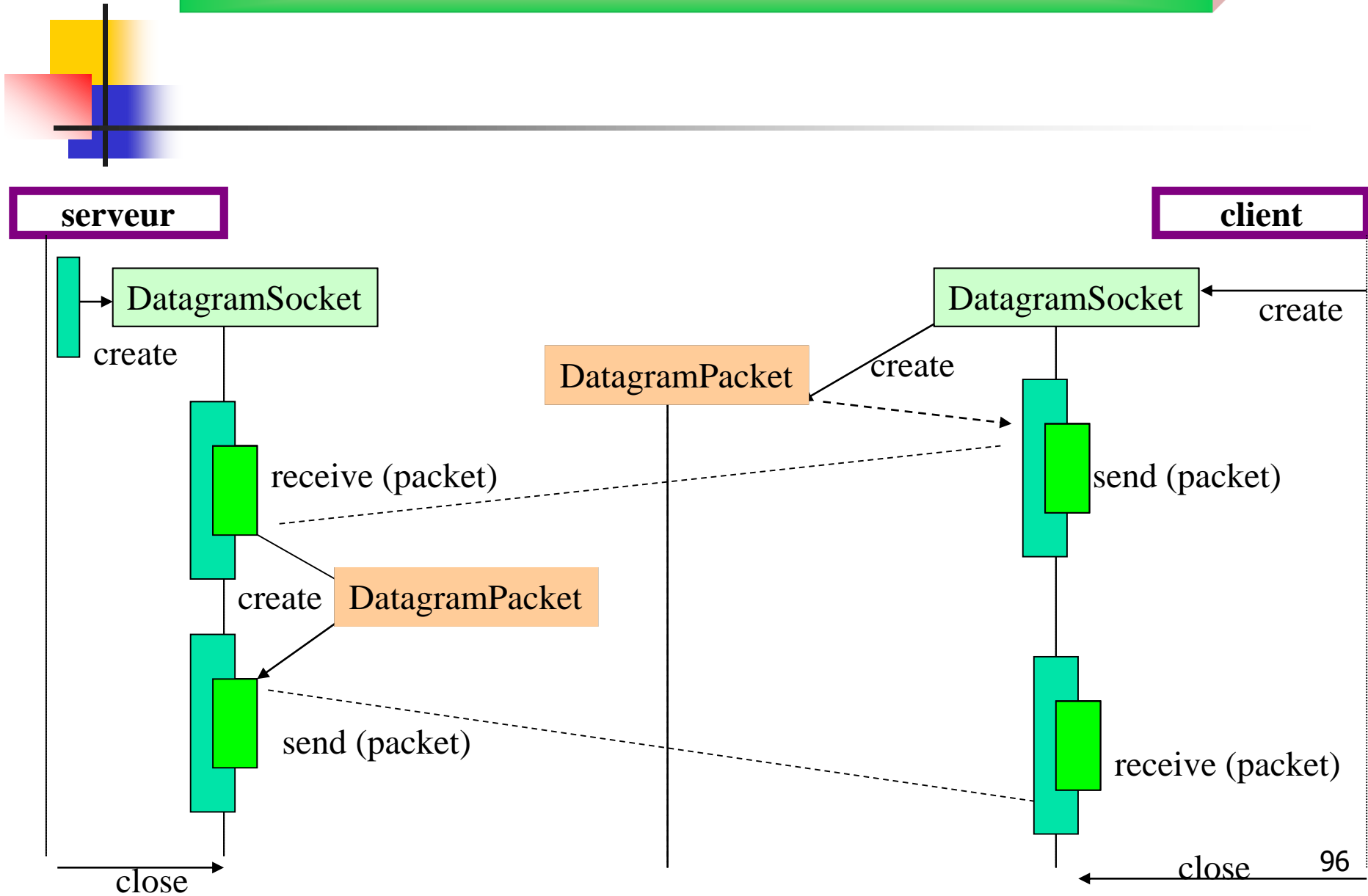
Émission et réception des datagrammes

Réception: `public void receive (DatagramPacket dp) throws IOException`

Cette méthode convertit le datagramme UDP reçu en objet *DatagramPacket*. A l'instar de la méthode `accept ()` de la classe *ServerSocket*, `receive ()` fige l'exécution du programme (code) en **attendant l'arrivée d'un datagramme**. Si votre programme doit faire d'autres tâches pendant ce temps, il faut alors appeler `receive ()` dans un thread séparé.

La méthode `getLocalPort ()` d'un *DatagramSocket* renvoie un entier représentant le numéro du port local sur lequel le socket courant écoute.

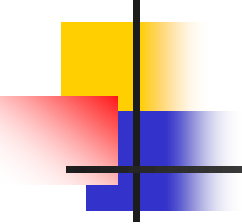
Communication client-serveur



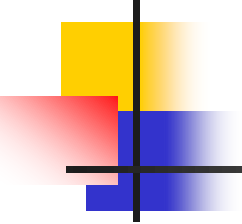
Exemple de client-serveur UDP

Envoie et réception d'une chaîne de caractères entre le client et le serveur.

```
package ndong.sockets.udp;
public class SocketUDPServer {
    public static void main (String[ ] args) throws java.net.SocketException, java.io.IOException {
        java.net.DatagramSocket socket = new java.net.DatagramSocket (4000);
        byte sendbuffer [ ] = null;
        byte receivebuff [ ] = new byte [256];
        java.net.DatagramPacket receivepacket = new java.net.DatagramPacket
                                                    (receivebuff, receivebuff.length);
        java.net.DatagramPacket sendpacket = null;
        System.out.println ("Serveur UDP prêt . ");
        socket.receive (receivepacket); //on recupere les packets client
        //on affiche les données reçues en octets en les convertissant en chaînes
        System.out.println ("Données reçues : " + new String (receivepacket.getData()));
        //on envoie une reponse indiquant d'où provient l'appel
        java.net.InetAddress adr = receivepacket.getAddress ( );
```



```
int port = receivepacket.getPort ( );
sendbuffer = ("vous êtes connecté depuis la machine "+adr +"et sur le port "+port).getBytes();
sendpacket = new java.net.DatagramPacket (sendbuffer,sendbuffer.length,adr,port);
socket.send (sendpacket);
}
}
```



Coté client

```
package ndong.sockets.udp;

public class SocketUDPCient {

public static void main(String [ ] args) throws Exception {
java.net.InetAddress adr = java.net.InetAddress.getByName ("localhost");
java.net.DatagramSocket socket = new java.net.DatagramSocket( );
byte buf [ ]="Bonjour".getBytes ( );
java.net.DatagramPacket sendpacket = new java.net.DatagramPacket (buf,buf.length,adr,4000);
buf = new byte [256];
java.net.DatagramPacket receivepacket=new java.net.DatagramPacket(buf,buf.length,adr,4000);
socket.send (sendpacket);
socket.receive (receivepacket);
System.out.println ("Données reçues du serveur. "+new String (receivepacket.getData ( )));
}
}
```

Commentaires sur l'exemple

Au niveau du serveur:

Un serveur UDP commence par ouvrir une socket sur un port donné, ici 4000.

On utilise ensuite deux tableaux d'octets pour l'envoi et la réception. Mais on est obligé de dimensionner le tampon de réception.

Comme la donnée reçue est une chaîne (mais reçue en octets), on la convertit en chaîne (new String (...)).

Le serveur envoie les données sur une adresse spécifique que l'on trouve sur le message reçu. Le numéro de port alloué au client est dynamique.

Au niveau client:

Le client ouvre une socket et envoie un texte en binaire. Il faut remarquer que seul le paquet comporte l'adresse du serveur et non la socket, et que la taille du paquet d'envoi est ajustée à la taille de l'information envoyée.

Multicast

Il est possible d'utiliser des adresses multicast permettant la diffusion simultanée d'une Information à plusieurs destinataires. Il existe des plages d'adresse multicast. On utilise également des datagrammes. Tous les clients actifs au moment de l'émission et qui écoutent sur cette adresse recevront simultanément la même information. La diffusion est sélective d'où le nom de multicast qui s'oppose à broadcast (diffusion).

Ce qu'il faut faire côté serveur:

```
DatagramSocket socket = new DatagramSocket ( );  
byte buf [ ] = "Bonjour clients ".getBytes ( );  
InetAddress group = InetAddress.getByName ( "230.0.0.2" );  
DatagramPacket receivepacket=new DatagramPacket (buf, buf.length, group,4000);
```

Adresse multicast

Multicast

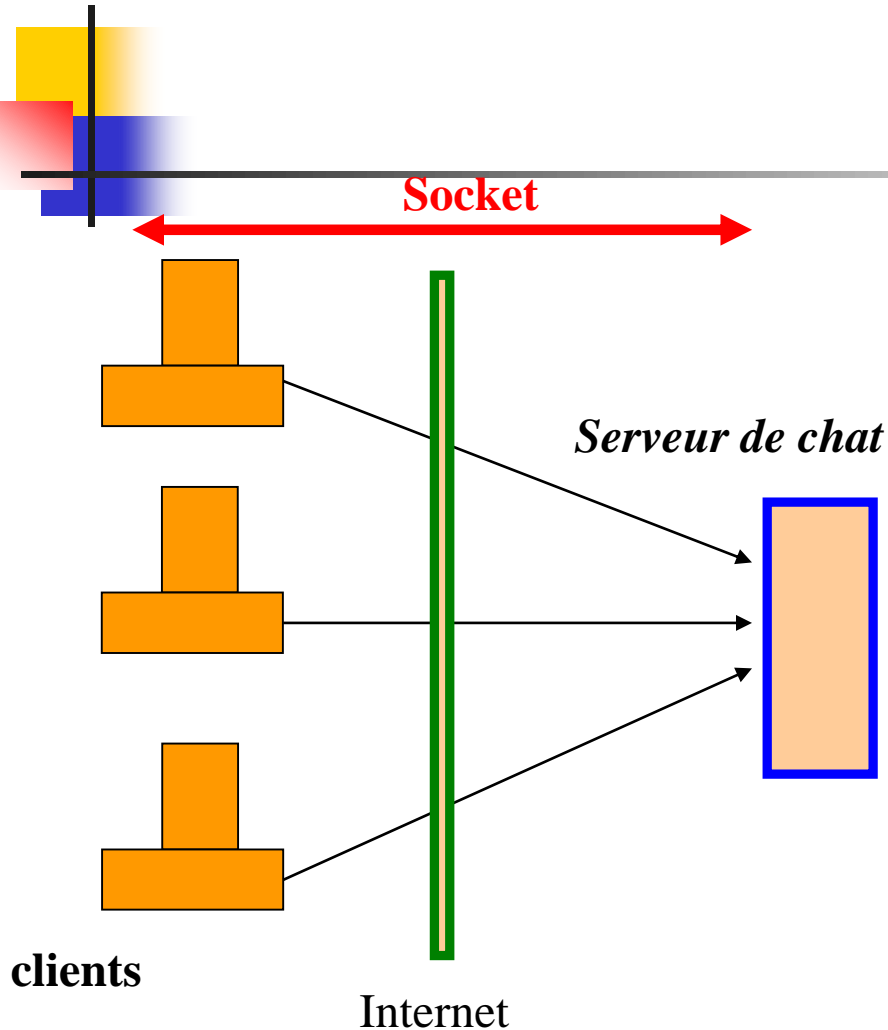
Côté client:

```
InetAddress adr = InetAddress.getByName ("230.0.0.2");  
InetSocketAddress address = new InetSocketAddress (adr,4000);  
MulticastSocket socket = new MulticastSocket (4000);  
socket.joinGroup (adr);  
byte buf [ ] = new byte [256];  
DatagramPacket packet = new DatagramPacket (buf, buf.length);  
socket.receive (packet);  
System.out.println (new String (packet.getData ( )));
```

Le multicast est aussi utilisé pour le clustering de serveurs ou la réplication de bases de données.

FIN

Exercice avec Sockets



Réaliser un **client/serveur multi-threadé** permettant au client d'envoyer des chaînes à un serveur qui les inverse et les retourne au client. Un client donné peut envoyer autant de messages qu'il souhaite. Et lorsqu'un client donné envoie un message **VIDE** au serveur, ce dernier termine et clôt la connexion avec ce client. Le serveur peut traiter plusieurs clients simultanément. Chaque fois qu'un client envoie une requête le serveur devra afficher l'adresse IP de ce client et le port par lequel le client a envoyé sa requête.

Essayer d'abord un client/serveur **CONSOLE** puis une version avec IHM (Swing, JavaFX).

Esquisse de solution

```
public class TCPserver extends Thread{
protected BufferedReader in;    protected PrintWriter out;    private Socket soc;
String s;
public TCPserver(Socket soc) {
    try {
        this.soc = soc;
        this.in = new BufferedReader(new InputStreamReader(this.soc.getInputStream()));
        this.out = new PrintWriter(this.soc.getOutputStream(), true);
    }
    catch (Exception e) {System.out.println ( "Pb lors de l'ecoute"); e.printStackTrace() ; }
}
public void run(){
    try{
        try{
            traitement();
            Thread.sleep(1000);
        }
        catch(InterruptedException e){
            System.out.println("Interruption grave du processus");
        }
    }
    catch(Exception er){}
}
```



```

public String reception ( ){
    // code manquant: compléter

}

public void traitement ( ) {
    // code manquant: compléter
}

public static void main (String args [ ])throws Exception{
    java.net.ServerSocket ss = new java.net.ServerSocket(2000);
    System.out.println("Demarrage serveur sur le port "+ss.getLocalPort ( ) + "....");
    while (true){
        Socket s=ss.accept ( );
        System.out.println("Traitement...avec le client"+s.getRemoteSocketAddress ( ));
        new TCPserver (s).start ( );
    }

}

}

```

```

public class TCPClientThread extends Thread {
    Socket s;
    Scanner entree=null;
    BufferedReader sin =null;
    PrintWriter sout =null;
    public TCPClientThread ( ){
    public void run ( ){
try {
    String line;
    while (true) {
        s = new Socket ("localhost",2000);
        Thread.sleep(1000);
        System.out.println ("Connected to " + s.getInetAddress ( ) + " on port " + s.getPort( ));
        sin = new BufferedReader (new InputStreamReader(s.getInputStream ( )));
        sout = new PrintWriter (s.getOutputStream ( ), true);
        System.out.print ("[Client Ask] > > >");
        entree = new Scanner (System.in);//pour saisir au clavier
        line = entree.nextLine ();//la ligne saisie au clavier

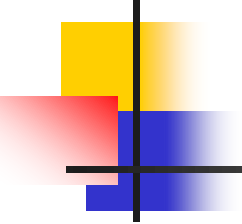
```

```

if(line.equals(""))
{
    s.close() ;
    System.out.println("Client s'arrete");
    break;
}
else {  sout.println(line);
        sout.flush() ;

        line = sin.readLine();
        System.out.println("recu du serveur: " + line);
    }
}
}
}
catch(Exception dd){
    System.out.println("probleme d'enfrmissement du Thread...");
}
finally{
    try{// dans tous les cas on ecrase le socket, security impose
        s.close();
    }
    catch(Exception er){ }
} }

```



```
public static void main(String[] args) {  
    new TCPClientThread ().start();  
}  
}
```



Module 3

RMI: Remote Method Invocation

(Invocation de Méthodes Distantes)

RMI est une API Java permettant de manipuler des *objets distants* (c'est-à-dire un objet instancié sur une autre machine virtuelle, éventuellement sur une autre machine du réseau) de manière transparente pour l'utilisateur, c'est-à-dire de la même façon que si l'objet était sur la machine virtuelle (JVM) de la machine locale.

Le but de RMI est donc 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. Ainsi un serveur permet à un client d'**invoquer des méthodes à distance** sur un objet qu'il instancie.

Deux machines virtuelles sont donc nécessaires (une sur le serveur et une sur le client) et l'ensemble des communications se fait en Java.

RMI permet donc de faire de la programmation d'applications distribuées.

Les classes de cette API sont disponibles dans le paquet **java.rmi**.



Concepts généraux de RMI

*Pour communiquer grâce à RMI, chaque objet Java d'une machine doit implanter une **interface** spécifiant lesquelles de ses méthodes sont accessibles aux clients. Ceux-ci peuvent dès lors appeler lesdites méthodes comme si elles existaient localement.*

Tous les détails sont cachés à l'utilisateur, qui accèdent aux objets et aux méthodes distantes comme s'ils résidaient sur sa propre machine.

*Le programme client (demandant l'appel de méthodes distantes) doit simplement importer le paquet et rechercher l'objet dans un **registre** et intercepter les **RemoteException** éventuellement déclenchées par les méthodes de l'objet, qui peut alors être manipulé comme tout objet résidant localement.*



Architecture répartie?

Une architecture **répartie** (*distributed*) se définit par des données et des tâches différentes sur les machines.

Les applications réparties ne partagent pas de mémoire, contrairement aux applications multi-processeur. Le problème reste alors la transmission des données entre tâches et la synchronisation entre celles-ci.

Il peut y avoir plusieurs raisons pour utiliser des applications réparties:

- répartir les données **géographiquement** ou **topologiquement** (*distributed applications*)
- permettre la **redondance** d'une application afin de diminuer le *taux de pannes* et augmenter la *fiabilité*.
- permettre la **montée en charge** (*scalability*),
- intégrer des applications existantes qui cohabitent dans l'entreprise.



Pourquoi les applications réparties?

La répartition est un état de fait pour un nombre important d'applications.

- **Besoins propres des applications**
 - Intégration d'applications existantes initialement séparées
 - Intégration massive de ressources
 - *Ex. gestion de données*
 - Nouvelles applications de l'informatique
 - *Informatique omniprésente : téléphones portables, capteurs, PDAs, ...*
- **Possibilités techniques**
 - *"On le fait parce qu'on peut le faire"*

Un peu de terminologie

Répartition = distribution

Système distribué

Application répartie



Caractéristiques des systèmes distribués

Définition d'un système distribué

- Ensemble composé d'éléments reliés par un système de communication
Les éléments ont des fonctions de traitement (processeurs), de stockage (mémoire, disques), de relation avec le monde extérieur
- Collaboration entre les différents éléments



Difficultés dans les systèmes distribués (1/2)

Interconnexions réseau



Latence

- Communiquer prend du temps
- Le temps est non déterministe
- Réactivité de l'application ?



Congestion

- On n'est pas tous seuls sur le réseau...



Pannes

Nombre d'entités impliqués

- Probabilité de panne proportionnelle au nombre d'entité impliquées
- Échelle mondiale : la panne n'est plus une exception mais une règle



Difficultés dans les systèmes distribués (2/2)

Dynamisme

La composition d'un système distribué change en permanence

- Ex. clients consultant un serveur web

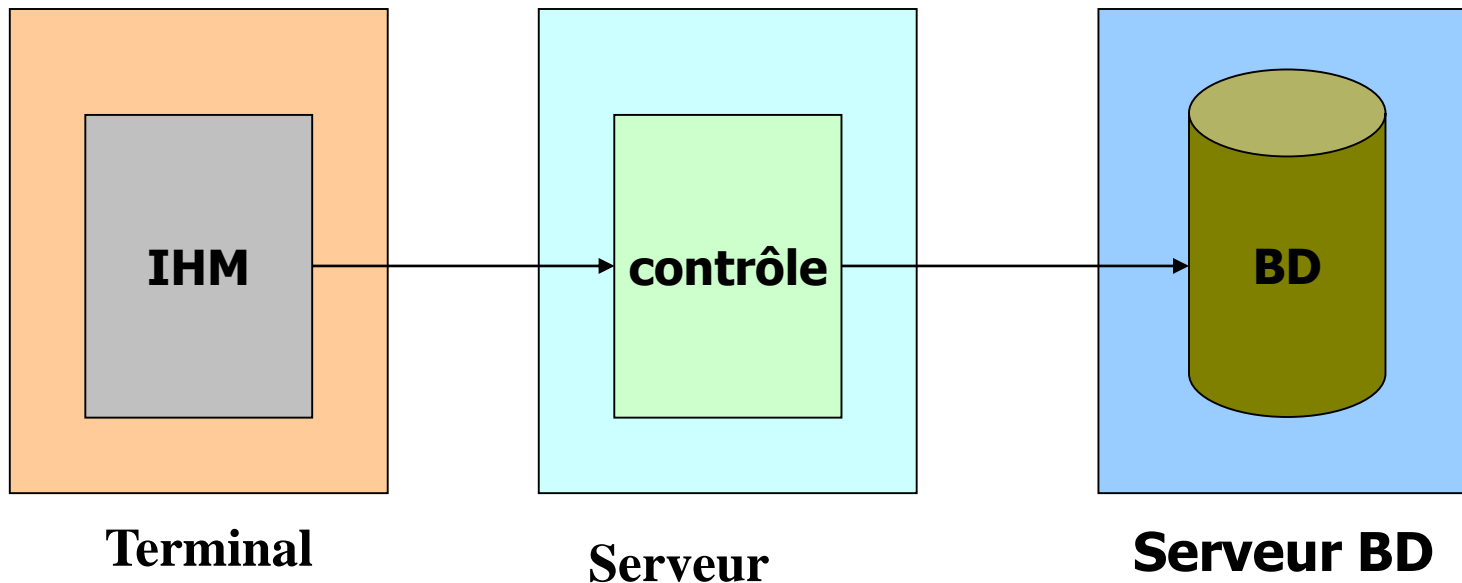
État global ?

**Si la structure ne change pas, on pourrait
construire un état global au bout d'un temps fini**

Administration

Qu'est ce qui est distribué?

- Une **application** = code, données, présentation
- Nombreuses possibilités de répartition
- Comment la mettre en oeuvre?





Besoins des systèmes distribués

- Le système doit pouvoir fonctionner (au moins de façon dégradée) même en cas de **défaillance** de certains de ses éléments
- Le système doit pouvoir résister à des **perturbations du système de communication** (perte de messages, déconnexion temporaire, performances dégradées)
- Le système doit pouvoir résister à **des attaques contre sa sécurité**



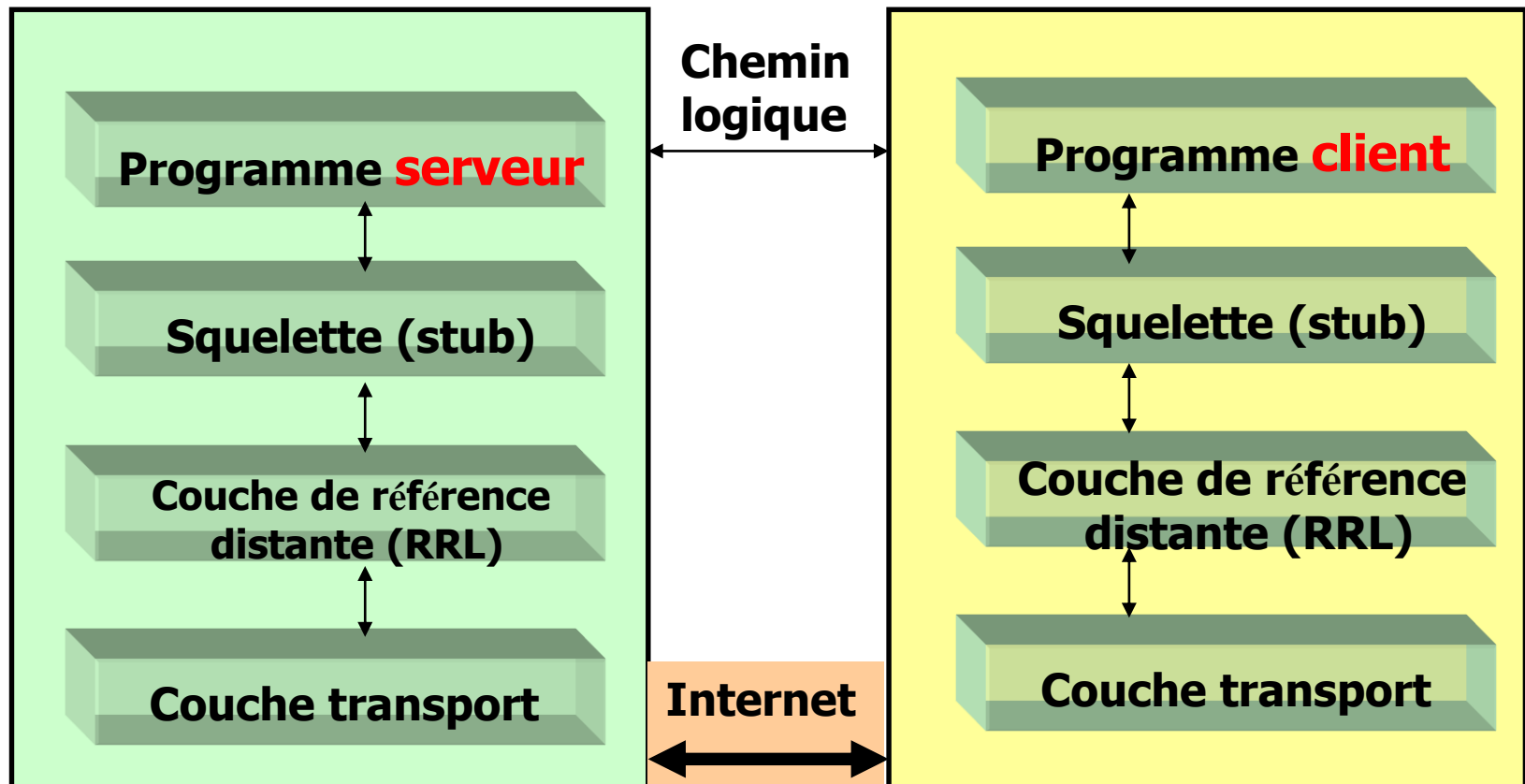
Distinction entre système et application

- **Système** : gestion de ressources communes, infrastructure
 - SE
 - Système de communication
 - Caractéristiques communes : cacher la complexité du matériel et des communications, fournissent des services d'un niveau d'abstraction plus haut
- **Application** : réponse à un problème spécifique, fournit des services à des utilisateurs (qui peuvent être des applications). Utilise les services fournis par le système.

Distinction non toujours évidente.

Architecture de RMI

(RMI fonctionne selon un modèle à souches schématisé ci-dessous.)



RRL= Remote Reference Layer = couche de référence distante

RMI: détails des opérations (1/3)

(Grands principes de fonctionnement)

La différence fondamentale qui existe entre un **objet distant** d'un **objet local** est que tous deux résident dans des **machines virtuelles distinctes**.

C'est le **passage par référence** qui permet de *définir les paramètres d'un objet* ou *d'accéder à ses valeurs*.

Les objets non **sérialisables** ne peuvent être passés aux méthodes distantes.

Le client communique avec le serveur via un **stub**, qui transmet les données reçues à la **couche de référence distante** laquelle interagit avec la **couche transport**.

La *couche transport* du client expédie les informations via **l'Internet** à la couche similaire du serveur, qui les communique à la *couche de référence distante* du serveur.

Les données parviennent alors au serveur par l'exécution d'un programme spécifique appelé *squelette* (*skeleton*).

Dans le sens contraire, le flux est tout simplement inversé.

Pour localiser le serveur, le client analyse un **registre** doté d'une méthode **lookup ()**, qui recherche puis télécharge le stub nécessaire à l'objet recherché.



RMI: détails des opérations (2/3)

*Le **stub** inclut le prototype de chacune des méthodes exportées par l'objet distant.*

Le client appelle donc les méthodes du stub et non de l'objet distant.

*En clair, le **stub** se substitue localement aux objets et aux méthodes du serveur, et lorsqu'ils sont sollicités, transmettent l'appel à la couche de référence distante.*

***La couche de référence distante** repose sur un protocole de référence distante spécifique indépendant du stub du client et du squelette du serveur; elle est chargée d'interpréter la référence distante. Selon le cas, celle-ci se rapporte à plusieurs machines virtuelles sur différents hôtes, ou bien à une seule machine virtuelle sur l'hôte local ou le système distant.*

En substance, la couche de référence distante traduit la référence locale au stub en référence distante à l'objet existant sur le serveur, et ensuite les données sont transmises à la couche transport.

RMI: détails des opérations (3/3)

La couche transport s'occupe de la **transmission** des données via *l'Internet* et assure la **configuration** et la **gestion de la connexion**, ainsi que la **localisation** et la **distribution** des objets distants.

Sur le serveur, cette couche **guette les connexions ou les données** et transmet chaque appel reçu à la *couche de référence distante* du serveur. Celle-ci convertit les références distantes expédiées par le client en références locales puis passe la requête au *squelette*, qui lit les paramètres et envoie les informations au logiciel serveur.

Ce dernier appelle la méthode requise, dont la sortie éventuelle effectue ensuite le trajet inverse depuis le *squelette* est les *couches de références distantes* et de *transport* du serveur jusqu'à *l'Internet* puis jusqu'aux couches de *transport*, de *référence distante*, puis de *stub* sur la machine cliente.



Notes

Les connexions et les transferts de données dans RMI sont effectués par Java sur TCP/IP (les packages *java.net.Socket* et *java.net.ServerSocket* assurent implicitement cette fonction) 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.

La transmission de données se fait à travers un système de couches, basées sur le modèle OSI afin de garantir une interopérabilité entre les programmes et les versions de Java.

La couche de référence (*RRL, Remote Reference Layer*) est chargée du système de localisation afin de fournir un moyen aux objets d'obtenir une référence à l'objet distant. Elle est assurée par le package *java.rmi.Naming* . On l'appelle généralement **registre RMI** car elle référence les objets.



Gestion de la sécurité: Fichier **java.policy**

En développement réseau en Java, il faut absolument gérer la sécurité sur l'accès aux données. Grâce au fichier *java.policy*, vous pouvez définir les politiques de sécurité sur l'accès aux données et les droits des utilisateurs.

- Le fichier *java.policy* par défaut est situé à
{java.home}/jre/lib /security /java.policy (***)
- Un usager peut avoir son propre fichier *java.policy* qui doit être situé dans son "*home directory*" (propriété *user.home*). Le fichier devra alors s'appeler
{user.home}/.java.policy
- Le fichier par défaut est lu en premier, suivi par le fichier usager

Remarque:

- 1) il faudra définir les nouveaux droits et politiques de sécurité dans ce fichier (à l'aide de la commande **grant**).
- 2) Si l'utilisateur définit son propre fichier de sécurité; il faut alors l'indiquer dans le fichier **java.security** défini dans (***)

Fichier **java.policy**

Le fichier *java.policy* contient deux types d'information:

- une instruction qui lui dit où aller chercher les clés privées

keystore "some_keystore_url", "keystore_type";

- zéro, une ou plusieurs instructions "grant":

grant [SignedBy "signer_names"] [, CodeBase "URL"]

{

permission permission_class_name ["target_name"]

[, "action"] [, SignedBy "signer_names"];

permission ...

};

Fichier `java.policy`: exemples

- Accorde à tout le monde la permission de lire le fichier ".tmp" (indépendamment du signataire, i.e. code signé ou pas, et de la source du code):

```
grant {  
    permission java.io.FilePermission ".tmp", "read";  
};
```

- Accorde deux permissions au code signé par Joe *ET* Loum:

```
grant signedBy "Joe, Loum" {  
    permission java.io.FilePermission "/tmp/*", "read";  
    permission java.util.PropertyPermission "user.*";  
}
```



Configuration minimale du fichier **java.policy**

Pour configurer le fichier **java.policy** afin que RMI puisse fonctionner, il faut ouvrir ce fichier avec un éditeur de texte, localiser la ligne suivante

// allows anyone to listen on un-privileged ports

permission java.net.SocketPermission "localhost:1024-", "listen";

et remplacer la par celle-ci

permission java.net.SocketPermission "*:1024-65535", "listen,accept,connect";



Résumé des opérations entre le client et le serveur

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*).
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 "*déséréalise*" 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.



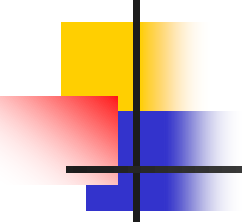
Mise en œuvre de RMI

(Création et manipulation d'objets distants)

L'essentiel des méthodes nécessaires à la production d'objets distants est réparti dans quatre paquets: `java.rmi`, `java.rmi.server`, `java.rmi.registry` et `java.rmi.dgc`.

- Le paquet `java.rmi` contient les **classes**, **interfaces** et **exceptions** qui concernent le client et permettent d'élaborer des programmes capables d'accéder à des objets distants sans constituer eux-mêmes de telles entités.
- Le paquet `java.rmi.server` détient les **classes**, **interfaces** et **exceptions** qui concernent le serveur et servent à créer des objets distants à l'intention des clients.
- Les **classes**, **interfaces** et **exceptions** du paquet `java.rmi.registry` s'utilisent pour localiser et nommer les objets distants.
- Le paquet `java.rmi.dgc` s'acquitte de la destruction d'objets inutiles.

Mise en œuvre de RMI: côté serveur



Créer un objet distant exige de définir au préalable une interface basée sur `java.rmi.Remote`. *Celle-ci ne possède aucune méthode*, son unique objectif consiste à permettre l'identification des objets distants.

Partant, on admet qu'un *objet distant se présente comme une instance* d'une classe implantant l'interface Remote, ou de toute autre interface basée sur Remote.

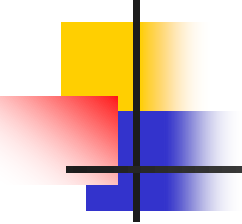
Les méthodes de l'objet distant accessibles aux objets clients dépendent de la sous-classe de Remote mise en œuvre.

Même si l'objet distant est doté de nombreuses méthodes publiques, **seules celles déclarées dans une interface distante sont mises à la disposition des clients**.

La majorité des *exceptions* déclenchées lors de l'invocation de méthodes à distance sont définies dans la classe `java.rmi.RemoteException` et chaque méthode de cette sous-interface doit par conséquent signaler l'éventualité d'une RemoteException.

Le programme n'a en principe aucun moyen de contrôler ces exceptions, qui ont trait pour la plupart au comportement des systèmes et à l'état des réseaux.

Mise en œuvre de RMI: côté serveur



La deuxième étape requiert de définir une classe implantant cette interface distante et basée sur `java.rmi.UnicastRemoteObject` .

Cette classe offre un ensemble de méthodes qui autorisent l'invocation de méthodes à distance.

Chaque objet distant nouvellement créé consigne lui-même son **existence et son nom** dans le **regsitre** grâce aux méthodes **bind** ou **unbind**.

Les **clients** pourront alors requérir *cet objet par son nom*, ou bien obtenir une liste de tous les objets existants.

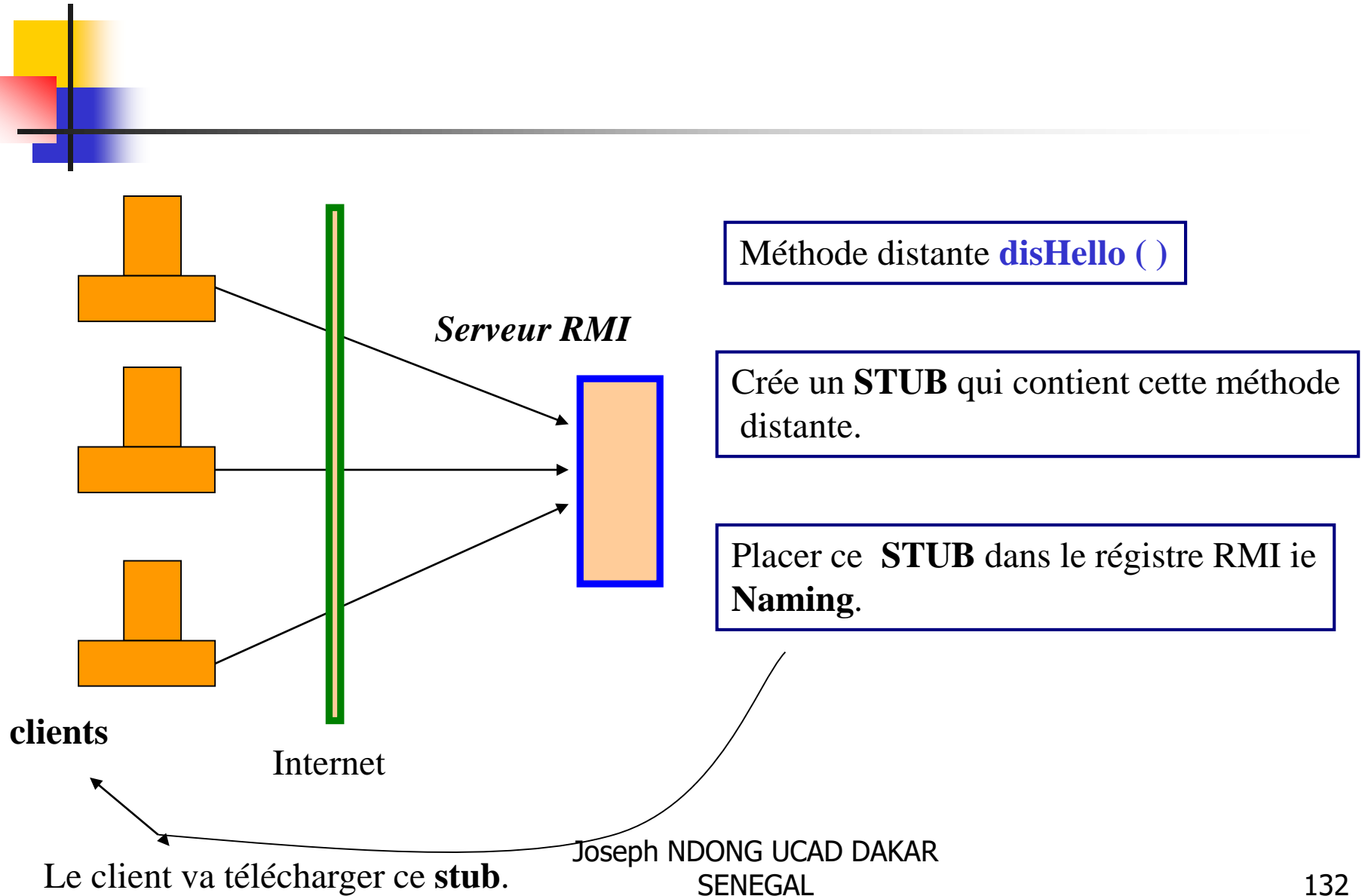
A cet instant, le codage de l'objet distant est terminé.

Mais avant de traiter les appels distants, il faut encore générer les stubs et les squelettes nécessaires au programme.

On rappelle que le **stub** contient les informations de l'nterface **Remote** et qu'un **squelette** abrite ces mêmes données sur le serveur.

Appliqué au(x) fichier(s) **.class**, l'utilitaire **rmic** délivre automatiquement **stubs** et **squelettes** à partir du code source de l'objet distant Java.

Exemple

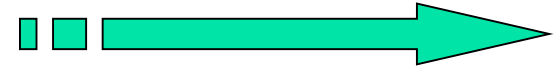




Exemple de mise en œuvre de RMI: coté serveur (1/4)

Cet exemple met en place une interface très simple destinée à un objet distant « **Hello Word** ». Son unique méthode, **disHello ()**, retourne une chaîne et provoque une **RemoteException** lorsque nécessaire.

Comment mettre en œuvre ce service ?





Exemple de mise en œuvre de RMI: coté serveur (2/4)

Étape 1:

On doit créer une interface spécifiant les méthodes accessibles à distance aux clients:

```
package test;

public interface Hello extends java.rmi.Remote {

    /*la seule méthode qu'on pourra invoquer à distance*/

    public String disHello ( ) throws java.rmi.RemoteException ;
}
```

Exemple de mise en œuvre de RMI: coté serveur (3/4)

Étape 2:

La deuxième étape requiert de définir une classe implantant cette interface distante et basée sur `java.rmi.UnicastRemoteObject`. Dans cette classe, on met les définitions des méthodes de l'objet distant (autrement dit les services offerts aux clients).

```
package test;
import java.rmi.*; import java.rmi.server.UnicastRemoteObject;
import java.io.*;
public class HelloImpl extends UnicastRemoteObject implements Hello, Serializable {
/*il est nécessaire et fondamentale de créer toujours ce constructeur*/
public HelloImpl () throws RemoteException {
    super ();
}
/*redéfinition de la seule méthode qu on pourra invoquer à distance*/
public String disHello ( ) throws java.rmi.RemoteException {

    return "Salut chers clients: "; } }
```

Attention , le constructeur ne sera pas exploitable
(appelable) par le client.

Exemple de mise en œuvre de RMI: coté serveur (4/4)

Étape 3:

Dans cette étape on crée un *objet distant* que tout client pourra bien invoquer.

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

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

        try {
            out .println ("démarrage serveur...");
            HelloImpl objetdistant = new HelloImpl ();
            Naming.rebind ("rmi://localhost:1099/NomObjetDistant", objetdistant);
            out .println ("serveur prêt");
        }
        catch (Exception e){err .println ("ERREUR: "+e.getMessage ());}
    }
}
```

Le client accède à l'objet distant grâce à ce nom



Notes importantes

Le constructeur **HelloImpl ()** appelle le *constructeur par défaut de la classe mère*.

Ceci est indispensable car la création d'un objet distant suppose de stipuler l'éventualité d'une **RemoteException**, si on sait que le constructeur de la super classe ne lance pas une telle exception.

Dans la classe Serveur, il s'agit de créer un objet distant et de lui associer un nom (ici le nom est "**NomObjetDistant**") dans *le registre de nommage* **java.rmi.Naming**, qui mémorise les noms des objets mis à la disposition par le serveur RMI.

Cet enregistrement se fait grâce aux méthodes **bind ()** ou **unbind ()**.

Une fois que l'objet est enregistré, le serveur indique qu'il est prêt à accepter les appels distants.

L'appel de **Naming.bind** nécessite que le registre fonctionne dans un autre processus de l'ordinateur.

Les clients pourront ensuite *requérir cet objet par son nom*.

ATTENTION: le client ne peut manipuler aucune méthode non définie dans l'interface distante.

Les objets non sérialisables (comme JDBC) ne peuvent pas être utilisés dans l'objet distant.



Génération des stub et skeleton Et lancement du serveur (1/2)

Maintenant que le codage est terminé côté serveur, avant de traiter les appels distants il faut générer les *stubs* et les *squelettes* nécessaires au programme.

Nous avons précisé plus haut qu'un stub contient les informations de l'interface **Remote** (dans cet exemple un objet et sa méthode **disHello ()**) et qu'un squelette abrite ces mêmes données sur le serveur.

Appliqué au fichier **.class**, l'utilitaire **rmic** délivre automatiquement le stub à partir du byte code obtenu avec le code source de l'objet distant.

Voici donc ce qu'il faudra faire:

```
C:\> javac Hello.java
```

```
C:\> javac HelloImpl.java
```

```
C:\> rmic -v1.2 HelloImpl
```

Il faut toujours avoir l'habitude de placer
le fichiers **.class** du stub dans un répertoire du **CLASSPATH**

```
C:\> javac Serveur.java
```

Génération des stub et skeleton Et lancement du serveur (2/2)

Le serveur peut dès lors être lancé, une fois que stub et squelette construits.

*Mais avant ce lancement, il faut invoquer le registre **Naming** de la sorte:*

C:\> **start rmiregistry**

C:\> **java** Serveur

démarrage serveur...
serveur prêt

Le registre doit s'exécuter de préférence en tâche de fond. L'écoute s'effectue par défaut sur le port **1099**.

Maintenant essayons de voir le développement côté client.

Compilation et exécution sous Eclipse

Supposons que vous travaillez sous un IDE comme Eclipse et que votre projet RMI se nomme **projetrmi** et qu'il se trouve dans un *workspace* nommé **D:\mesprojets**.

Voici concrètement ce qu'il faut faire:

- 1) Mettre le répertoire **D:\mesprojets\projetrmi\bin** dans le CLASSPATH du système.
- 2) **D:\mesprojets\projetrmi\>javac -d .\bin .\src\test\Hello.java**
- 3) **D:\mesprojets\projetrmi\>javac -d .\bin -classpath .\bin .\src\test\HelloImpl.java**
- 4) **D:\mesprojets\projetrmi\>javac -d .\bin -classpath .\bin .\src\test\Serveur.java**
- 5) Génération du stub sous MS-DOS:
D:\mesprojets\projetrmi\>rmic -v1.2 -d .\bin -classpath .\bin test.HelloImpl
- 6) Démarrage du registre RMI sur le port par défaut 1099
C:\> start rmiregistry
- 7) Démarrage du serveur
D:\mesprojets\projetrmi\>java -classpath .\bin test.Serveur

Mise en œuvre de RMI: coté client



Pour être en mesure d'appeler une méthode distante, le client doit en premier lieu extraire une référence distante à l'objet visé.

Pour ce faire, il lui faut appeler la méthode **lookup (String URL)** du service d'enregistrement.

La procédure de nommage est propre à chaque registre: pour sa part la classe **Naming** implantée dans **java.rmi** localise les objets grâce à des URL similaires aux URL HTTP.

La chaîne **rmi** indique le protocole employé et implique que l'URL référence un objet distant.

Le champ de l'URL consacré au fichier contient quant à lui le nom de l'objet.

Les champs destinés au nom de l'hôte et au numéro de port demeurent eux inchangés.

Fondamental: à l'instar des objets stockés dans des **Hashtable** ou **Vector**, contenant des objets de classe hétérogènes, *tout objet extrait d'un registre perd l'énoncé de son type.*

Par conséquent, avant de manipuler un objet, **il faut impérativement le convertir vers l'interface distance** qu'il implante (et non la véritable classe, auquel le client n'a pas accès).

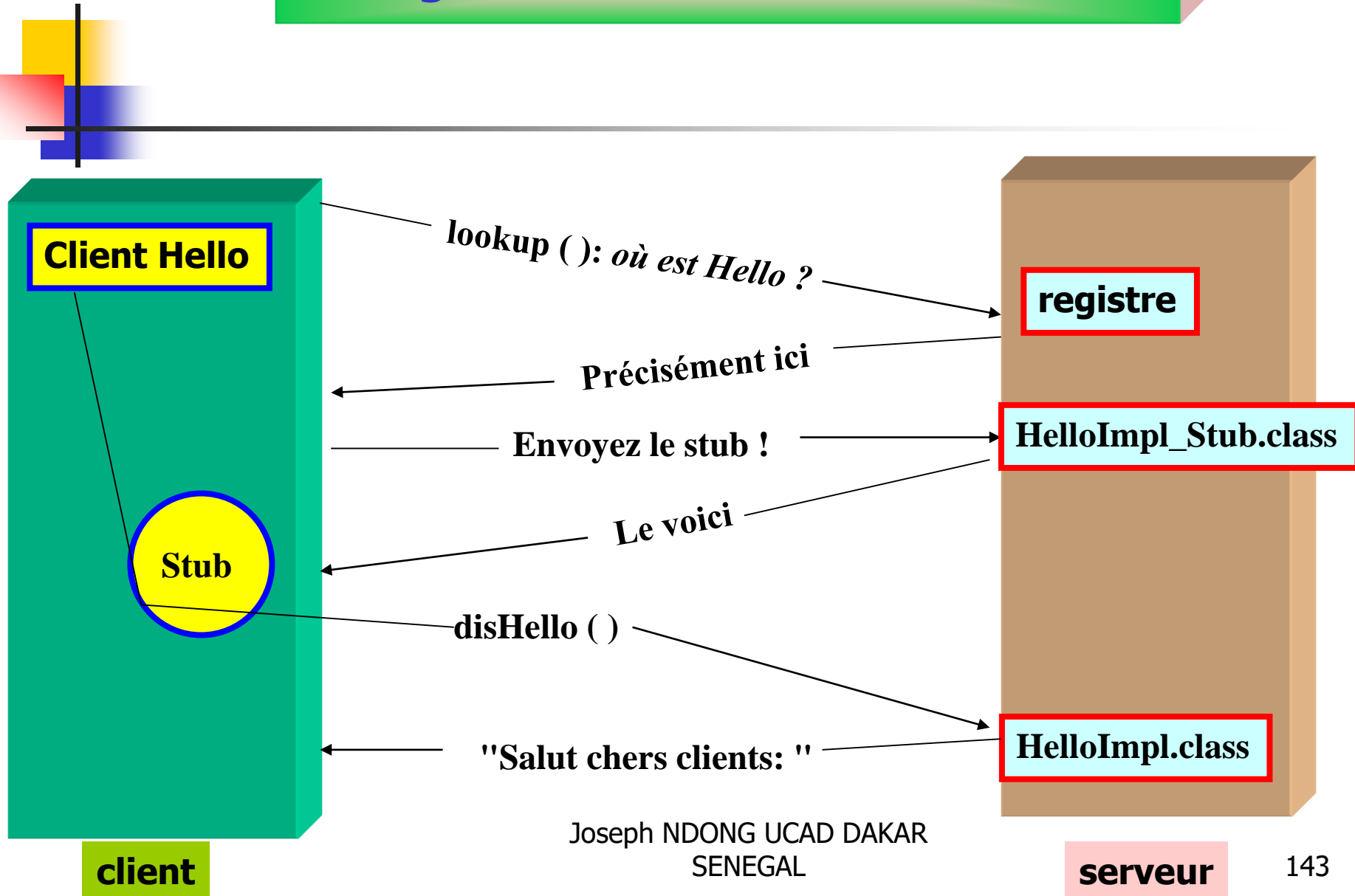
Mise en œuvre de RMI: coté client

Une fois que l'objet est extrait et converti, le client peut alors faire appel à aux méthodes distantes comme s'il s'agissait de méthodes d'un objet local.

```
import java.rmi.*;
public class Client {
    public static void main (String [ ] args) {
        System.setSecurityManager (new RMISecurityManager ( ));
        try {
            Hello sh = (Hello) java.rmi.Naming.lookup ("rmi://localhost /NomObjetDistant");
            String s = sh.disHello ( ); // appel de la methode distante
            System.out .println (s) ;
        }
        catch (Exception e){ e.printStackTrace ( ) ;}
    }
}
```

Le registre,
le serveur et le client
résident sur le même hôte.

Dialogue entre le client et le serveur



Compilation et lancement du client

La compilation et l'exécution du programme client se fait comme de façon standard.

C:\> **javac** Client.java

C:\> **java** Client

Salut chers clients:

Remarques importantes et pratiques:

Les commandes **javac**, **java** et **rmic** sont lancées ici de façon simple et standard. En *pratique* il faut tenir compte du *package* contenant les classes et de la variable CLASSPATH.

- ✓ Souvent vous aurez à séparer le serveur du client dans deux *packages différents*. Dans ce cas, il faudra *jarer* le serveur et l'importer au niveau du package client afin de pouvoir importer les classes nécessaires à la compilation du client.
- ✓ Avec la commande **javac**, il est souhaitable de *toujours* déposer les byte codes (**.class**) dans un répertoire *classes* que vous créerez dans votre *répertoire principal de travail*. Cela se fera avec l'option **-d** de cette même commande.
- ✓ Souvent aussi, l'option **-classpath** est nécessaire lorsque la compilation ou l'exécution exige de situer l'emplacement des classes requises.

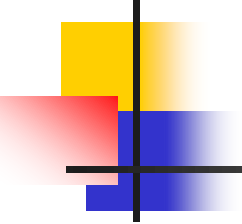


Résumé de la mise en œuvre du client et du serveur

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 *ServerSocket*, permettant la communication par protocole TCP)
3. créer les classes pour le stub et le squelette grâce à la commande *rmic*
4. 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.
5. Créer un programme client capable d'accéder aux méthodes d'un objet sur le serveur grâce à la méthode *Naming.lookup ()*
6. Compiler l'application cliente
7. Instancier le client et lancer son exécution

Changement de port



Comme on peut le constater, RMI fonctionne par défaut sur le port **1099**.
Il serait préférable, par mesure de sécurité, de modifier ce port et d'en prendre un qui n'est pas connu d'avance des utilisateurs.

//Au niveau serveur j'utilise le port 7512

```
Naming.rebind ("rmi://localhost:7512/NomObjetDistant", objetdistant) ;
```

//Le client aussi devra specifier ce même port

```
Hello sh = (Hello) java.rmi.Naming.lookup ("rmi://localhost:7512/NomObjetDistant");
```

Lors du démarrage du registre, il faut obligatoirement mentionner le port comme suit:

```
start rmiregistry 7512
```

Démarrage du **registre** dans le **code du serveur**



Au lieu de démarrer le **registre** en *mode ligne de commande*, on peut préférer le démarrer dans le code même du serveur, il suffit de faire

```
java.rmi.registry.Registry reg = java.rmi.registry.LocateRegistry.createRegistry (14445);  
reg.rebind ("rmi://localhost:14445/ndong", objetdistant);
```

Et le client pourra accéder à ce registre en faisant:

```
Registry r = LocateRegistry.getRegistry(14445);  
ICalcul i=(ICalcul) r.lookup("rmi://localhost:14445/ndong");
```



Activation des objets du serveur

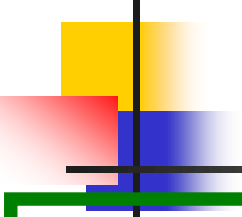
Dans l'exemple précédent, nous avons instancié et placé des objets dans le registre RMI sans être sûr qu'ils seront utilisés. Cette façon de faire peut être dégradant aussi dans le cas où l'on a un **très grand nombre d'objets**. Il serait préférable d'activer les objets serveur seulement au moment où un client donné sollicite les fonctions proposées.

Le mécanisme d'activation permet de retarder la construction de l'objet distant. L'objet ne sera construit que si le client le sollicite.

Pour réaliser cette activation au besoin, le code du client reste complètement inchangé, vous ne modifiez que le code du serveur.

Voici les différentes étapes du processus d'activation

Modifications au niveau de la classe de l'objet distant



```
package essai;
import java.rmi.MarshalledObject;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationID;

public class IHelloImpl extends Activatable implements IHello, java.io.Serializable{

    private String name;
    private static final long serialVersionUID = 6092320152170302524L;
    public IHelloImpl (ActivationID id, MarshalledObject data) throws Exception{
        Super (id,0);           // le 0 pour assigner un PORT adapté au port de l'écouteur
        this.name = (String)data.get ( ); // désérialise les informations de construction de l'objet
    }
    public String disHello ( ) throws java.rmi.RemoteException{
        return "Salut chers clients: ";
    }
}}
```

Modifications au niveau de la classe du serveur

```
package essai;

import java.io.File; import java.rmi.MarshalledObject; import java.rmi.Naming;
import java.rmi.activation.Activatable; import java.rmi.activation.ActivationDesc;
import java.rmi.activation.ActivationGroup; import java.rmi.activation.ActivationGroupID;
import java.rmi.activation.ActivationGroupDesc; import java.util.Properties;

public class IHelloActivator {
    public static void main(String [ ] args) throws Exception{
        Properties p = new Properties();
        p.put("java.security.policy", new File("server.policy").getCanonicalPath());
        ActivationGroupDesc group = new ActivationGroupDesc(p,null);
        ActivationGroupID id=ActivationGroup.getSystem().registerGroup(group);
        MarshalledObject param = new MarshalledObject("nomdactivation");
        String classDir=".";
        String classURL = new File(classDir).getCanonicalFile().toURI().toString();
        ActivationDesc de = new ActivationDesc(id,"essai.IHelloImpl",classURL,param);
        IHello ii=(IHello)Activatable.register(de);
        Naming.rebind("rmi://localhost:1099/ndong", ii);
        System.out.println(" Serveur PRET");}}}
```



Explications

Étant donné que la construction de l'objet est retardée, elle doit se faire de façon standardisée.

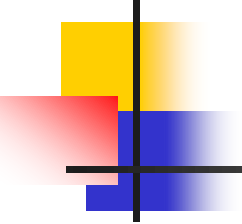
Donc on doit fournir au constructeur de l'objet distant deux paramètres:

- Un **ID d'activation** (que l'on passe au constructeur de la super classe),
- un seul objet contenant toutes les informations de construction, emballées dans un **MarshaledObject**. Dans notre cas, il s'agit simplement du nom de l'objet (champ **name**).

Dans le programme d'activation:

- Il faut d'abord définir un **groupe d'activation** qui décrit les paramètres ordinaires de lancement de la JVM hébergeant les objets du serveur. Les paramètres les plus importants sont les règles de sécurité. Au lieu d'utiliser le fichier **java.policy** comme dans le premier cas, je propose de créer un fichier de sécurité **server.policy** donnant toute permission.

Je place ce fichier dans le répertoire racine de mon application.



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

server.policy

Je construis le **descripteur de groupe d'activation** en mettant:

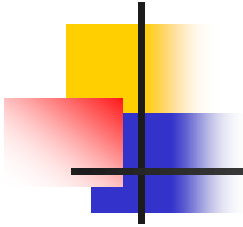
```
Properties p = new Properties();
p.put("java.security.policy", new File("server.policy").getCanonicalPath());
ActivationGroupDesc group = new ActivationGroupDesc(p, null);
```

Ensuite je crée un **ID de groupe** avec l'instruction:

```
ActivationGroupID id=ActivationGroup.getSystem().registerGroup(group);
```

Maintenant, pour chaque objet distant, il faut construire son **descripteur d'activation**, comme suit:

```
MarshaledObject param = new MarshaledObject("nomdactivation");
String classDir=".";
String classURL = new File(classDir).getCanonicalFile().toURI().toString();
ActivationDesc de = new ActivationDesc(id, "essai.IHelloImpl", classURL, param);
```

On passe ensuite le descripteur créé à la méthode **Activation.register** qui renvoie un objet d'une classe qui implémente les interfaces distantes de la classe d'implémentation. C'est cet objet qui sera enregistré dans le registre:

```
IHello ii=(IHello)Activatable.register(de);  
Naming.rebind("rmi://localhost:1099/ndong", ii);
```

A la différence du premier exemple, le programme d'activation se ferme après enregistrement et liaison des récepteurs d'activation. Les objets du serveur sont uniquement créés lors du premier appel à la méthode.



Lancement du programme serveur

- 1) Compiler les fichiers source
- 2) Générer le **STUB** avec la commande **rmic**.
- 3) Lancer le registre RMI
- 4) Démarrer le démon d'activation RMI nommé *rmid*

Créer la fichier **rmid.policy** et placez le dans le répertoire courant de votre projet

```
grant
{
permission com.sun.rmi.rmid.ExecPermission "${java.home}${/}bin${/}java";
permission com.sun.rmi.rmid.ExecOptionPermission "-Djava.security.policy=*";
};
```

rmid.policy

```
start rmid -J-Djava.security.policy=rmid.policy
```

- 5) Lancer le programme principal



Prospection du paquet **java.rmi**

Le paquet **java.rmi** contient les classes visibles des clients (c à d les objets appelant les méthodes distantes). Clients comme serveurs doivent importer ce paquet, qui satisfait tous les besoins des premiers et une partie de ceux des seconds et se compose d'une **interface**, de **deux classes** et d'un **ensemble d'exceptions**.



L'interface `java.rmi.Remote`

L'interface **Remote** permet d'identifier les objets distants.

Dépourvue de méthodes, **celles-ci sont d'ordinaire déclarées dans une sous-classe de cette interface implantée par l'objet** et constituent alors *les seules méthodes de l'objet accessibles à distance*.

Un objet peut posséder plusieurs interfaces *Remote*. En principe, une méthode déclarée dans une telle interface est toujours accessible au client, mais il faut noter que ce dernier ne peut convertir l'objet vers plus d'une interface à la fois.

C'est pourquoi, plutôt que de séparer les interfaces de l'objet, nous préconisons de les réunir dans une interface composite puis d'exploiter celle-ci.



La classe java.rmi.Naming

Cette classe représente un registre comparable à une sorte de DNS pour objets.

Ses entrées sont formées d'un **nom** et d'une **référence à un objet**.

Lorsqu'un client transmet un URL tel que **rmi://fst.info.edu /monObjetDistant**, par exemple, **il obtient en retour une référence à l'objet distant correspondant**.

Naming est le seul registre RMI disponible à nos jours, mais d'autres devraient voir le jour dans l'avenir.

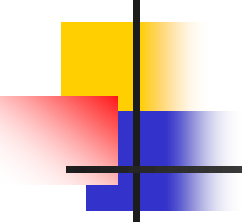
Cette classe présente néanmoins un défaut en ce qu'elle exige le client à connaître le nom du serveur de résidence de l'objet.

Cette classe possède cinq méthodes: **bind ()**, qui associe un nom à un objet distant donné; **list ()** qui renvoie la liste de tous les noms consignés dans le *registre*; **lookup ()**, qui recherche un objet distant donné d'après son URL; **rebind ()**, qui associe le nom fourni à un objet distant; **unbind ()** qui supprime un nom du registre.



```
public static Remote lookup (String url)
throws RemoteException, NotBoundException,
AccessException, UnknownHostException
```

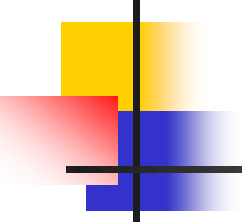
Cette méthode permet au client d'extraire une interface Remote associée à l'objet mentionné dans l' URL. Avec **rmi://fst.info.edu /monObjetDistant** par exemple, la méthode **lookup ()** retournera l'objet **monObjetDistant** de **fst.info.edu** . Elle déclenche une **NotBoundException** si le nom indiqué n'est associé à aucun objet, une **AccessException** si le client n'est pas habilité à accéder au *registre*. Une **UnknownHostException** signale que l'hôte spécifié dans l' URL n'a pas pu être localisé.



```
public static void bind (String url, Remote obj)
throws RemoteException, AlreadyBoundException,
AccesException, UnknownHostException
```

Un serveur emploie la méthode **bind ()** afin de lier un nom (*monObjetDistant* par ex.) à un objet distant. Sous réserve que l'opération réussisse, les clients peuvent ensuite extraire cet objet du registre grâce à l'URL *url*.

L'échec de la recherche de l'hôte se solve par une **UnknownHostException** tandis q'une **AccesException** indique que le client n'est pas autorisé à affecter des noms aux objets dans le registre. Enfin une **AlreadyBoundException** dénote que l'URL est déjà associée à un objet et une **RemoteException** sanctionne un dépassement de temps (registre non localisé).



```
public static void unbind (String url)
throws RemoteException, NotBoundException,
AccesException, UnknownHostException
```

La méthode **unbind ()** supprime du registre l'objet correspondant à l' URL fourni. Pendant de **bind ()**, elle annule le résultat de cette dernière.

Une **NotBoundException** signale que l' URL passé n'est lié à aucun objet existant. Les autres exceptions sont identiques à celles levées par **bind ()**.

```
public static void rebind (String url, Remote obj)
throws RemoteException, AccesException,
UnknownHostException
```

Identique à la méthode **bind ()**, **rebind ()** associe cependant l' URL à l'objet même si l' URL identifie déjà un autre objet. L'URL identifie alors dorénavant le nouvel objet. par conséquent **rebind ()** ne renvoie pas **d'AlreadyBoundException** mais des **RemoteException**, **AccesException** et **UnknownHostException** dans les mêmes circonstances que **bind ()**.



```
public static String [ ] list (String url)  
throws RemoteException,  AccesException,  
UnknownHostException
```

Cette méthode communique une liste (tableau de chaînes), de l'ensemble des URL référençant des objets. Le paramètre url remplace l'énoncé de l'URL du registre *Naming* à traiter.

Seuls le protocole, l'hôte et le port sont analysés, la section indiquant le nom de l'objet est quant à elle ignorée.



La classe `java.rmi.RMISecurityManager`

Le **stub** extrait par le client peut provenir d'un serveur douteux. En cela, le stub est au client ce que l'applet est au logiciel de navigation, c'est-à-dire une entité à surveiller étroitement.

Bien que les pouvoirs du stub se limitent à convertir en binaires les paramètres et effectuer l'opération inverse sur les valeurs de retour, puis à les transmettre sur le réseau, la machine virtuelle le considère comme toute autre classe téléchargée, dont les méthodes peuvent se révéler destructrices.

Les *stubs* générés par **rmic** sont inoffensifs mais un stub artisanal peut éventuellement tenter de lire des fichiers ou de supprimer de données.

La machine virtuelle Java n'autorise le téléchargement de stubs qu'à condition d'installer un objet **SecurityManager** (comme toute autre classe, un stub peut aussi être extrait du CLASSPATH local) dont le rôle est analogue à celui de l'*AppletSecurityManager* standard qui surveille les applets.



La classe `java.rmi.RMISecurityManager`

Les applications recourent au **`RMISecurityManager`** pour se protéger des stubs nuisibles. Ce gestionnaire de sécurité très simple (dérivé de *`java.lang.SecurityManager`*) **préserve uniquement la définition de la classe et permet simplement d'accéder aux objets distants, de passer des paramètres à une méthode distante ou d'en retourner une valeur.**

En l'absence d'un gestionnaire de sécurité, **seuls les stubs des fichiers locaux du `CLASSPATH` sont accessibles**, si bien que le **téléchargement de stubs et de classes depuis le réseau et toujours assujetti à la présence d'un tel gestionnaire.**



Le constructeur: `public RMISecurityManager ()`

La classe **RMISecurityManager** dispose d'un unique constructeur sans paramètre. La méthode *statique* `System.setSecurityManager ()` permet d'initialiser le gestionnaire de sécurité.

Le nouveau *SecurityManager* est le plus souvent créé directement comme ceci:

```
System.setSecurityManager (new RMISecurityManager ( ));
```



Le paquet: `java.rmi.registry`

Le client localise l'objet distant sur le serveur grâce à une référence, qu'il recherche dans le registre du serveur. Ce *registre* permet aux client de se renseigner sur la disponibilité des objets distants et d'en obtenir les références.

Le registre `java.rmi.Naming` implante l'interface `java.rmi.registry.Registry` et toutes ses méthodes publiques en sont héritées.

Grâce à l'interface `Registry` et à la classes `LocateRegistry`, le client peut extraire les objets distants par leur nom.

On rappelle que l'interface `Registry` offre les cinq méthodes, `bind ()`, `list ()`, `unbind ()`, `lookup ()` et `rebind ()`, vues précédemment.

L'interface `Naming` dispose d'un champ statique final, `REGISTRY_PORT`, indiquant le numéro du port par défaut sur lequel le registre écoute, soit le port **1099**.

Ce numéro de port peut être changé en faisant:

```
java.rmi.registry.Registry reg = java.rmi.registry.LocateRegistry.createRegistry (14445);
```

La classe: LocateRegistry

La classe `java.rmi.Registry.LocateRegistry` permet au client de localiser le service d'enregistrement grâce à quatre versions de la méthode statique `getRegistry ()`.

*/*sur le port par défaut, 1099, de l'hôte local */*

`public static Registry getRegistry () throws RemoteException`

*/*sur le port spécifié de l'hôte local */*

`public static Registry getRegistry (int port) throws RemoteException`

*/*sur le port par défaut, 1099, de l'hôte spécifié */*

`public static Registry getRegistry (String hote) throws RemoteException,
UnknownHostException`

*/*sur le port spécifié de l'hôte spécifié */*

`public static Registry getRegistry (String hote, int port) throws RemoteException
UnknownHostException`

Si la chaîne hôte est « **null** », `getRegistry ()` effectue la recherche sur l'hôte local, et sur le port par défaut si le paramètre port est négalif.



La classe: **LocateRegistry**

Ainsi, si par exemple, le nom de l'objet distant est « **NomObj** » sur la machine **hote.serveurmi.com**, on peut faire la recherche à distance comme suit:

```
Registry r = LocateRegistry.getRegistry ("hote.serveurmi.com");
```

```
InterfaceObjDistant iod = (InterfaceObjDistant) r.lookup ("NomObj");
```

Et à partir de l'objet **iod** représentant l'objet distant, on peut faire appel à n'importe quelle méthode de l'interface distante comme:

```
iod.appelMethodeDistante ( );
```



Le paquet: `java.rmi.server`

Ce package contient l'ensemble des outils nécessaires à la création d'objets distants. Il est utilisé par tous les objets dont les méthodes sont accessibles au client. Les classe essentielles de ce paquet sont `RemoteObject`, qui est la base de tous les objets distants, `RemoteServer`, fondée sur la précédente et `UnicastRemoteObject` (dont la plupart de vos classes seront probablement basées).

La classe `UnicastRemoteObject` gère la *sérialisation* et la *désérialisation* des paramètres et des valeurs de retour. Un objet `UnicastRemoteObject` existe sur une seule machine et communique grâce à des sockets TCP. Rien de ceci ne nécessite l'intervention du programmeur.

Les clients qui appellent des méthodes distantes et ne sont pas eux-mêmes des objets distants n'utilisent pas les classes de ce paquetage, donc inutile d'importer le paquet.



La classe: `java.rmi.server.RemoteObject`

Cette classe est une version spéciale de `java.lang.Object` dédiée aux objets distants et ses méthodes `toString()`, `hashCode()`, `clone()` et `equals()` sont spécialement conçues dans cette optique.

`equals()` compare les références de deux `RemoteObject` et renvoie « **true** » si toutes deux pointent vers *le même objet distant*.

`toString()` renvoie une chaîne décrivant l'objet distant. Cette méthode identifie **le nom de l'hôte**, le **numéro de port de provenance** de l'objet *distant* et fournit une *référence*.

`hashCode()` fait correspondre à chaque objet un entier unique, qui peut être utilisé comme valeur de hachage. La valeur retournée est la même pour toutes les références à un même objet.



La classe: `java.rmi.server.RemoteServer`

Basée sur *RemoteObject*, cette classe abstraite permet de créer des serveurs distants du type *UnicastRemoteObject*. Cette dernière est la seule sous classe de *RemoteServer* prédéfinie dans le paquet, mais rien n'empêche d'en ajouter d'autres (serveur distant UDP ou multipoint, par exemple) en dérivant les sous classes adéquates.

Cette classe dispose de deux constructeurs:

`protected RemoteServer ()`

`protected RemoteServer (RemoteRef r)`

Vous n'instancierez jamais cette classe, mais plutôt, sa sous classe basée sur *UnicastRemoteObject* ou bien une autre de votre choix. Il vous faudra appeler un de ces constructeurs dans la première ligne de la définition du constructeur de la sous classe en question.

Obtenir des informations sur le client

La classe *RemoteServer* définit deux méthodes chargées de localiser le client:

/ renvoie une chaîne indiquant le nom de la machine cliente responsable de l'appel de la méthode exécutée à cet instant*/*

public static String getClientHost () throws *ServerNotActiveException*

*/*retourne un entier représentant le numéro du port de ce même client*/*

public static String getClientPort () throws *ServerNotActiveException*

Si aucune méthode distante ne s'exécute dans le thread courant, ces deux méthodes lancent une *ServerNotActiveException*.

Comment tracer les appels ?

A des fins de débogage, il peut être utile de visualiser les appels à l'objet distant et les réponses de celui-ci.

Pour obtenir un compte rendu complet, il suffit de passer un objet **OutputStream** à la méthode **setLog ()**:

```
public static void setLog (OutputStream os)
```

Si *os* vaut « **null** », la gestion des traces est désactivée.

Pour visualiser tous les appels à *System.err* (qui transmet le compte rendu à la console Java), par exemple, on notera: **RemoteServer.setLog (System.err)**;

La méthode **getLog ()** permet d'ajouter d'autres renseignements dans le compte rendu en plus de ceux fournis par la classe *RemoteServer*.

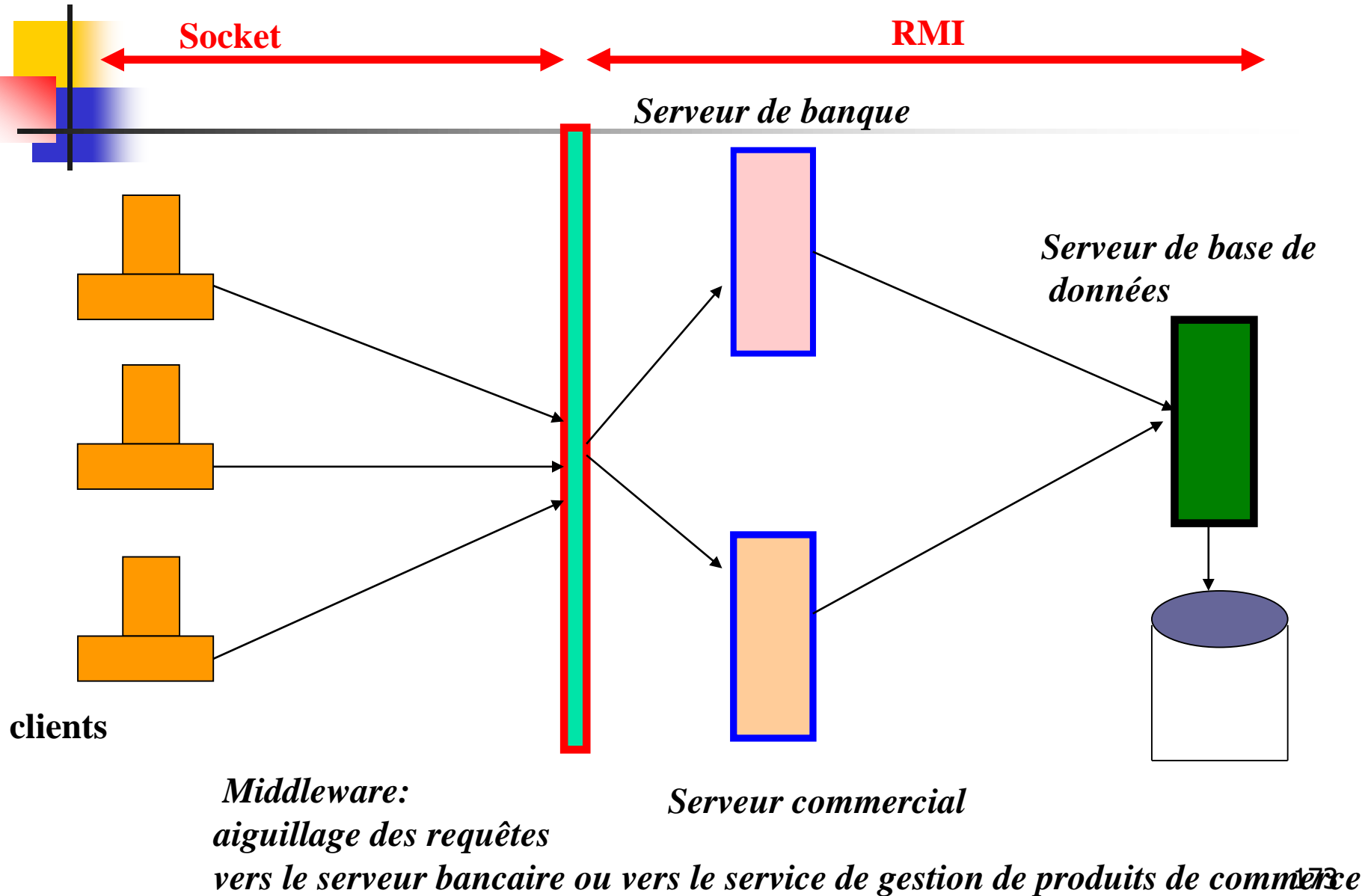
Une fois extrait le *PrintStream* du compte rendu grâce à:

```
public static PrintStream getLog ( ),      on peut ajouter des commentaires:
```

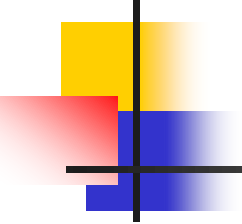
```
PrintStream p = RemoteServer.getLog ( );  
p.println ("l objet distant a été sollicité " +n + "fois au total");
```

FIN

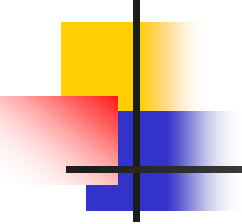
Exercice avec Sockets + RMI (1/2)



Exercice avec Sockets + RMI (2/2)

- 
- Le client est sous forme d'interface graphique et il est soit un client bancaire (il peut créer des comptes faire les opérations classiques sur un compte) soit un client commercial (gestion de produits alimentaires).
 - Le Middleware est un serveur qui s'interface entre le client et les deux serveurs bancaire et commercial. Il permet d'aiguiller les requêtes clientes vers le BON serveur.
 - Le serveur bancaire gère la gestion des comptes en banque
 - Le serveur commercial gère les produits de consommation
 - Le serveur de base de données gère le stockage des données de banque et de commerce.
 - La communication entre les clients et le Middleware se fait par sockets
 - La communication entre le Middleware et le reste se fait par RMI.

NB: vous devez utiliser des ordinateurs distincts pour héberger le Middleware, le serveur de banque, le serveur commercial et le serveur de base de données.



Exploiter les exercices proposés dans le cahier d'exercices pour bien asseoir les notions théoriques de base vues dans ce cours.

Module 4

La gestion dynamique des objets et l'introspection

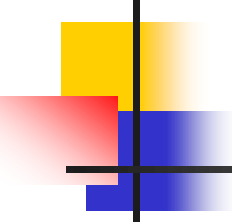
L'introspection (ou *réflexion*) est un mécanisme qui permet de connaître le contenu d'une classe dynamiquement. Il permet notamment de savoir ce que contient une classe sans en avoir les sources. Cette fonctionnalité est très utilisée avec les **JavaBeans**.

Les différentes classes utiles pour l'introspection sont rassemblées dans le package **java.lang.reflect**.

Nous allons montrer dans ce chapitre comment avoir de façon dynamique des informations sur une classe. On notera qu'il existe plusieurs façon de procéder

L'introspection est très utilisée par certains frameworks du monde JEE, comme Struts.

La classe `java.lang.Class`



Lorsqu'un programme Java est lancé, le système d'exécution de Java, gère, pour tous les objets, ce que l'on appelle « *l'identification de type à l'exécution* ». Cette info mémorise la classe à laquelle chaque objet appartient. L'info de type au moment de l'exécution est employée par la machine virtuelle pour sélectionner les méthodes à exécuter. Cette info est accédée grace à la classe ***Class***.

Les instances de la classe ***Class*** sont des objets représentant les classes du langage. Il y aura une instance représentant chaque classe utilisée : par exemple la classe *String*, la classe *Frame*, la classe *Class*, etc

Ces instances sont créés automatiquement par la *machine virtuelle* lors du chargement de la classe. Il est ainsi possible de connaître les caractéristiques d'une classe de façon dynamique en utilisant les méthodes de la classe ***Class***.

La classe Class permet :

- *de décrire une classe ou une interface par introspection* : obtenir son nom, sa classe mère, la liste de ces méthodes, de ses variables de classe, de ses constructeurs et variables d'instances, etc ...
- .
- *d'agir sur une classe en envoyant, à un objet Class des messages* comme à tout autre objet. Par exemple, créer dynamiquement à partir d'un objet *Class* une nouvelle instance de la classe représentée

La classe `java.lang.Class`

Comment obtenir une instance de la classe `Class`?

La classe `Class` ne possède *aucun constructeur public* mais il existe plusieurs façons d'obtenir un objet de la classe `Class`.

❖ Connaître la classe d'un objet:

La méthode `getClass ()` définit dans la classe `Object` renvoie une instance de la classe `Class`. Par héritage, tout objet java dispose de cette méthode.

```
String chaine = "test";  
Class cetteclasse = chaine.getClass ();  
System.out.println ("classe de l'objet chaine = "+ cetteclasse.getName ( ));
```

```
COMME SORTIE :      classe de l'objet chaine = java.lang.String
```

La classe `java.lang.Class`

❖ Obtenir un objet `Class` à partir d'un nom de classe:

La classe *Class* possède une méthode statique **forName ()** qui permet à partir d'une chaîne de caractères désignant une classe d'instancier un objet de cette classe et de renvoyer un objet de la classe *Class* pour cette classe.

Cette méthode peut lever l'exception **ClassNotFoundException**.

```
try {  
    Class classe = Class.forName ("java.lang.String");  
    System.out.println ("classe de l'objet chaine = " +classe.getName ( ));  
}  
catch (Exception e) {    e.printStackTrace ( );  
}
```

COMME SORTIE : classe de l'objet chaine = java.lang.String

La classe `java.lang.Class`

❖ Une troisième façon d'obtenir un objet `Class`:

Il est possible d'avoir un objet de la classe `Class` en écrivant `type.class` ou type est le **nom** d'une classe.

```
Class c = Object.class ;  
System.out.println ("classe de Object = " + c.getName ( ));
```

```
COMME SORTIE :      classe de l'objet = java.lang.Object
```

Les méthodes de la classe java.lang.Class

*La classe **Class** fournit de nombreuses méthodes pour obtenir des informations sur la classe qu'elle représente. Voici les principales méthodes :*

static Class forName (String):

Instancie un objet de la classe dont le nom est fourni en paramètre et renvoie un objet Class la représentant

Class [] getClasses ():

Renvoie les classes et interfaces publiques qui sont membres de la classe

Constructor [] getConstructors ():

Renvoie les constructeurs publics de la classe

Class [] getDeclaredClasses () : Renvoie un tableau des classes définies comme membre dans la classe

Constructor [] getDeclaredConstructors () :

Renvoie tous les constructeurs de la classe

Field [] getDeclaredFields ()

Renvoie un tableau de tous les attributs définis dans la classe

Method getDeclaredMethods ()

: Renvoie un tableau de toutes les méthodes

Field getFields ()

: Renvoie un tableau des attributs publics

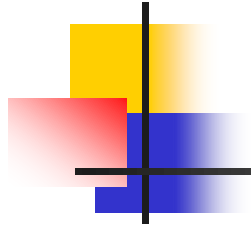
Class [] getInterfaces ()

: Renvoie un tableau des interfaces implémentées par la classe

Method getMethod ()

: Renvoie un tableau des méthodes publiques de la classe incluant celles héritées

Les méthodes de la classe java.lang.Class



`int getModifiers ()`: Renvoie un entier qu'il faut décoder pour connaître les modificateurs de la classe.

`package getPackage ()`:

Renvoie le package de la classe

`Classe getSuperClass ()`:

Renvoie la classe mère de la classe

`boolean isArray ()`:

Indique si la classe est un tableau

`boolean IsInterface ()`:

Indique si la classe est une interface

`Object newInstance ()`:

Permet de créer une nouvelle instance de la classe

Recherche d'informations sur une classe



*En utilisant les méthodes de la classe **Class**, il est possible d'obtenir quasiment toutes les informations sur une classe.*

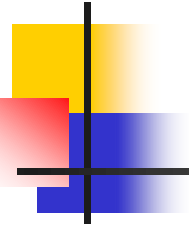
Comment connaître la super classe d'une classe ?

La classe **Class** possède une méthode **getSuperClass ()** qui retourne un objet de type **Class** représentant la *classe mère* si elle existe sinon elle retourne **null**.
Pour obtenir toute la hiérarchie d'une classe il suffit d'appeler successivement cette méthode sur l'objet qu'elle a retourné.

Comment trouver les modifieurs d'une classe ?

La classe **Class** possède une méthode **getModifiers ()** qui retourne un **entier** représentant les modificateurs de la classe.
Pour décoder cette valeur, la classe **Modifier** possède plusieurs méthodes qui attendent cet entier en paramètre et qui retourne un **booléen** selon leur fonction : *isPublic, isAbstract, isFinal*

La classe **Modifier** ne contient que des *constantes* et des *méthodes statiques* qui permettent de déterminer les modificateurs d'accès :



<code>boolean isAbstract(int):</code>	Renvoie true si le paramètre contient le modificateur abstract
<code>boolean isFinal(int):</code>	Renvoie true si le paramètre contient le modificateur final
<code>boolean isInterface (int):</code>	Renvoie true si le paramètre contient le modificateur interface
<code>boolean isNative (int):</code>	Renvoie true si le paramètre contient le modificateur native
<code>boolean isPrivate (int):</code>	Renvoie true si le paramètre contient le modificateur private
<code>boolean isProtected (int):</code>	Renvoie true si le paramètre contient le modificateur protected
<code>boolean isPublic (int):</code>	Renvoie true si le paramètre contient le modificateur public
<code>boolean isStatic (int) :</code>	Renvoie true si le paramètre contient le modificateur static
<code>boolean isSynchronized (int):</code>	Renvoie true si le paramètre contient le modificateur <i>synchronized</i>
<code>boolean isTransient (int):</code>	Renvoie true si le paramètre contient le modificateur transient
<code>boolean isVolatile (int):</code>	Renvoie true si le paramètre contient le modificateur volatile

```
Class classe = String.class ;  
int m = this.getClass( ).getModifiers ( );  
if (Modifier.isPublic(m))  
System.out.println ("public");
```

Joseph NDONG UCAD DAKAR
SENEGAL

Comment trouver les interfaces implémentées par une classe ?

La classe **Class** possède une méthode **getInterfaces ()** qui retourne un tableau d'objet de type **Class** contenant les interfaces implémentées par la classe.

Exemple:

```
public Vector getInterfaces ( ) {  
    Vector cp = new Vector ( );  
    Class [ ] interfaces = this.getClass( ).getInterfaces ( );  
    for (int i = 0; i < interfaces.length; i++) {  
        cp.add (interfaces[i].getName ( ));  
    }  
    return cp;  
}
```

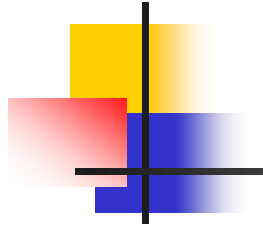
Comment trouver les champs publics d'une classe ?

La classe **Class** possède une méthode **getFields ()** qui retourne les attributs *public* de la classe.

Cette méthode retourne un tableau d'objet de type **Field**.

La classe **Class** possède aussi une méthode **getField (String nomChamp)** qui retourne un objet de type **Field** si *nomChamp* est un champ défini dans la classe ou dans une de ses classes mères. Si la classe ne contient pas d'attribut de nom *nomChamp*, cette méthode lève une exception de type *NoSuchFieldException*.

La classe **java.lang.reflect.Field** représente *un attribut d'une classe* ou d'une *interface* et permet d'obtenir des informations sur cet attribut. Elle possède plusieurs méthodes :



<code>String getName ():</code>	Retourne le nom de l'attribut
<code>Class getType ():</code>	Retourne un objet de type Class qui représente le type de l'attribut
<code>Class getDeclaringClass ():</code>	Retourne un objet de type Class qui représente la classe qui définit l'attribut
<code>int getModifiers ():</code>	Retourne un entier qui décrit les modificateurs d'accès. Pour les connaître précisément il faut utiliser les méthodes <i>static</i> de la classe <i>Modifier</i> .
<code>Object get (Object):</code>	Retourne la valeur de l'attribut pour l'instance de l'objet fourni en paramètre.

Il existe aussi plusieurs méthodes **getXXX ()** ou **XXX** représente un type primitif et qui renvoie la valeur dans ce type.

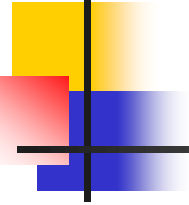


Exemple:

```
public java.util.ArrayList getChampsPublics ( ) {  
    java.util.ArrayList cp = new java.util.ArrayList ( );  
    java.lang.reflect.Field[] champs = this.getClass ( ).getFields ( );  
    for (int i = 0; i < champs.length; i++)  
        cp.add(champs[i].getType( ).getName()+" "+champs[i].getName( ));  
    return cp;  
}
```

Comment rechercher les constructeurs de la classe?

La classe **Class** possède une méthode **getConstructors ()** qui retourne un *tableau d'objet* de type **Constructor** contenant les constructeurs de la classe.



La classe **Constructor** représente un constructeur d'une classe. Elle possède plusieurs méthodes :

- String getName ():** Retourne le nom du constructeur
- Class [] getExceptionTypes ():** Retourne un tableau de type **Class** qui représente les exceptions qui peuvent être propagées par le constructeur
- Class [] getParametersType ():** Retourne un tableau de type **Class** qui représente les paramètres du constructeur
- int getModifiers ():** Retourne un entier qui décrit les modificateurs d'accès. Pour les connaître précisément il faut utiliser les méthodes *static* de la classe **Modifier**.
- Object newInstance (Object []):** Instancie un objet en utilisant le constructeur avec les paramètres fournis à la méthode



Exemple:

```
public java.util.ArrayList getConstructeurs ( ) {  
    java.util.ArrayList cp = new java.util.ArrayList ( );  
    java.lang.reflect.Constructor [ ] constructeurs = this.getClass() .getConstructors ( );  
    for (int i = 0; i < constructeurs.length; i++) {  
        cp.add (constructeurs [i].getName ( ));  
    }  
    return cp;  
}
```

Comment chercher les méthodes déclarées **public** d'une classe ?

Pour consulter les méthodes d'un objet, *il faut obtenir sa classe* et lui envoyer le message **getMethod ()**, qui *renvoie les méthodes publique* qui sont déclarées dans la classe et qui sont héritées des classes mères.

Elle renvoie un tableau d'instances de la classe **Method** du package *java.lang.reflect* .
*Une méthode est caractérisée par un **nom**, une **valeur de retour**, une **liste de paramètres**, une **liste d'exceptions** et une **classe d'appartenance**.*

La classe **Method** contient plusieurs méthodes :

Class [] getParameterTypes (): Renvoie un tableau de classes représentant les paramètres.

Class getReturnType (): Renvoie le type de la valeur de retour de la méthode.

String getName (): Renvoie le nom de la méthode

int getModifiers (): Renvoie un entier qui représentent les modificateur d'accès

Class [] getExceptionTypes (): Renvoie un tableau de classes contenant les exceptions propagées par la méthode

Class getDeclaringClass []: Renvoie la classe qui définit la méthode



Module 5

JavaBeans

Les java beans (ou *beans* simplement) sont des *composants logiciels réutilisables* introduits par le JDK 1.1.

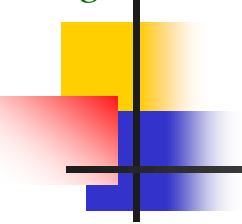
Les beans sont prévues pour pouvoir interagir avec d'autres beans au point de pouvoir développer une application simplement *par assemblage* de beans.

L'écriture d'un bean ne demande aucun concept ou élément syntaxique nouveau ; d'autre part, son utilisation ne requiert aucune bibliothèque ni extension particulière.

Les beans ne forment pas une hiérarchie de classes, ni même une API.

Pour créer des beans il faut se conformer simplement à certaines règles.

Les JavaBeans possèdent quelques caractéristiques fondamentales:



Règles de dénomination: les règles d'écriture et de « nommage » des méthodes doivent être respectées lorsqu'on écrit un bean. Elles permettent, de reconnaître et de manipuler les caractéristiques d'un bean (c.-à-d. ses propriétés(champs publics) , méthodes et événements), sans avoir à en connaître l'implémentation.

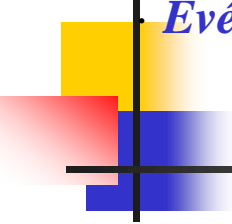
la persistance : elle permet grâce au mécanisme de sérialisation de sauvegarder l'état d'un bean pour le restaurer; ainsi si on assemble plusieurs beans pour former une application, on peut la sauvegarder.

la communication: grâce à des événements qui utilise le modèle des écouteurs introduit par java 1.1

l'introspection : ce mécanisme permet de découvrir de façon dynamique l'ensemble des éléments qui compose le bean (attributs, méthodes et événements) sans avoir le source. La réflexion permet à des beans qui se rencontrent durant l'exécution de faire presque tout ce qui aurait été possible s'ils avaient été assemblés durant la compilation.

la possibilité de paramétrer le composant : les données du paramétrage sont conservées dans des propriétés.

Les principaux éléments de Java que les beans mettent en oeuvre sont les suivants :

- 
- **Événements.** La communication entre beans (a priori développés indépendamment les uns des autres) est assurée par le mécanisme des événements. Une manière d'attacher deux beans *A* et *B* consiste à enregistrer *B* comme « auditeur » (*listener*) d'événements dont *A* est la source ; cela établit une voie de communication de *A* vers *B*.
 - **Propriétés.** L'état d'un bean est défini par les valeurs de ses *propriétés*. Chaque propriété se manifeste par un couple de méthodes *get<Prop>* et *set<Prop>*. Définir *B* comme valeur d'une propriété de *A* est une autre manière d'attacher deux beans *A* et *B*.
 - **Sérialisation.** Autant que possible, l'état d'un bean peut être sauvegardé (on dit *sérialisé*) dans un fichier, totalement ou partiellement, en vue de sa désérialisation ultérieure.
 - **Archivage.** Un bean se compose généralement de plusieurs classes. L'utilitaire **jar** permet de réunir celles-ci en un unique fichier, compressé, utilisable par la machine Java grâce à un « manifeste » qui en décrit le contenu.

Constructeur

Les beans doivent posséder un constructeur sans paramètre.

Celui ci devra initialiser l'état du bean avec des valeurs par défaut.

```
public class Entier implements java.io.Serializable {  
    private int valeur = 0;  
    public int getValeur () {  
        return valeur;  
    }  
    public void setValeur (int valeur) {  
        this.valeur = valeur;  
    }  
}
```

Le constructeur d'un bean n'est pas un élément très important.

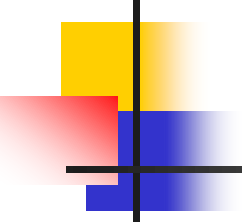
La « vraie » construction d'un bean ne se produit pas au sein de l'application qui utilise le bean, durant l'exécution, mais auparavant, dans l'outil de développement qui a servi à assembler l'application. Dans cet outil, un certain *éditeur de propriétés* aura permis de définir l'état du bean à partir d'un ensemble de valeurs utiles. A la fin du développement, le bean, dans l'état ainsi défini et ensemble avec les autres composants de l'application, aura été sérialisé dans un fichier.



Constructeur

Ultérieurement, quand l'application sera exécutée et que le bean deviendra nécessaire, il sera simplement *désérialisé* et il aura tout de suite les valeurs intéressantes définies durant le développement. Cette désérialisation d'un bean sérialisé, distincte de la construction ou *instanciation* habituelle, s'appelle *instanciation de bean*.

Évènements



La technologie **JavaBeans** utilise, sans ajout ni modification, le modèle événementiel de Java, introduit dans le langage lors de la parution de la version du jdk 1.1.

le *bean* est la source et les autres composants qui souhaitent être informé sont nommé « Listeners » ou « écouteurs » et doivent s'enregistrer auprès du bean qui maintient la liste des composants enregistrés.

Il est nécessaire de définir les méthodes qui vont permettre de gérer la liste des écouteurs désirant recevoir l'événement. Il faut définir deux méthodes :

- **public void addXXXListener (XXXListener li)** pour enregistrer l'écouteur li
- **public void removeXXXListener (XXXListener li)** pour enlever l'écouteur li de la liste

L'objet de type *XXXListener* doit obligatoirement implémenter l'interface **java.util.EventListener** et son nom doit terminer par le mot « Listener ».

Les événements sont regroupés en *catégories d'événements* .

Exemples, tirés de **awt** : **MouseEvent** (événements liés à la souris), **KeyEvent** (événements en rapport avec le clavier)

ActionEvent (qui contient un seul événement, signalant l'actionnement d'un bouton), etc.

Dire qu'un objet *O* peut être écouteur d'une catégorie *<C>* *d'événements* c'est dire que *O* possède toutes les méthodes par lesquelles les événements de *<C>* sont notifiés.

En Java, cela se fait en déclarant que *O* implémente une certaine interface, définie à cet effet. L'interface correspondant à une catégorie *<C>* **Event** se nomme *<C>Listener*.

Exemples : **MouseListener**, **KeyListener**, **ActionListener**, etc.

Les écouteurs d'un événement doivent être enregistrés auprès de la source de tels événements. Cela se fait par un appel de la méthode **add <C>Listener (<C>Listener X)**, qui incorpore *X* dans la liste des écouteurs à prévenir lorsque *<C>* se produit.

Exemples : **addMouseListener**, **addKeyListener**, **addActionListener**, etc.

Propriétés

Les propriétés contiennent des données qui gèrent l'état du composant : ils peuvent être de type primitif ou être un objet.

Il existe quatre types de propriétés :

- les propriétés *simples*
- les propriétés *indexées* (*indexed properties*)
- les propriétés *liées* (*bound properties*)
- les propriétés *liées avec contraintes* (*Constrained properties*)

Une propriété a un nom, <nomPropriété>, un type, <TypePropriété>, et se manifeste à travers deux méthodes :

```
public void set<nomPropriété>(<TypePropriété> valeur);
```

```
public <TypePropriété> get<nomPropriété>( );
```

Lorsque le type de la propriété est **boolean**, la méthode *get* peut prendre la forme :

```
public boolean is<nomPropriété>( );
```

La méthode *setPropriété* n'est pas obligatoire. On dit qu'une propriété existe dès que l'objet en question possède la méthode *getPropriété*. Si la méthode *setPropriété* n'existe pas on dit qu'il s'agit une propriété « en lecture seule ».

Les propriétés simples

Les propriétés sont des *variables d'instance* du bean qui possèdent des méthodes particulières pour **lire** (getter) et **modifier** (setter) leur valeur. L'accès à ces propriétés doit se faire via les méthodes getter.

```
private int longueur; //variable d'instance
/*méthode de lecture de la propriété longueur*/
public int getLongueur ( )
{
    return longueur;
}
/*pour la modification de la propriété longueur*/
public void setLongueur (int longueur)
{
    this.longueur = longueur;
}
```

Les propriétés indexées (indexed properties)

Les propriétés qui représentent des collections de valeurs de même type **<TypeElement>** ont des méthodes supplémentaires. D'une part il y a deux méthodes qui présentent la collection comme un tableau :

```
public <TypeElement> [ ] get <nomPropriete >( );  
public void set <nomPropriete >(<TypeElement>[ ] valeur);
```

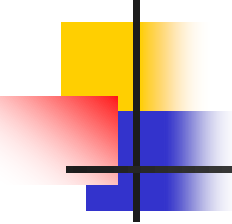
D'autre part il y a deux méthodes qui permettent d'atteindre un élément particulier de la collection :

```
public <TypeElement> get <nomPropriete >(int indice);  
public void set <nomPropriete >(int indice, <TypeElement> valeur);
```

```
private float [ ] notes = new float [5]; // propriété indexée notes  
/*Accès individuel aux éléments de la propriété indexée notes*/  
public float getNotes (int i ) {  
    return notes [i];  
}  
/*modification des éléments de la propriété indexée notes*/  
public void setNotes (int i ; float unenote) {  
    this.notes [i] = unenote;  
}
```

NB: Il est aussi possible de définir des méthodes « get » et « set » permettant de mettre à jour **tout** le tableau.

Les propriétés liées (Bound properties)

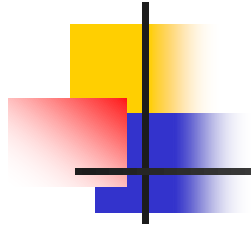


Il est possible d'informer d'autre composant du changement de la valeur d'une propriété d'un bean. Les java beans peuvent mettre en place un mécanisme qui permet pour une propriété d'enregistrer des composants qui seront informés du changement de la valeur de la propriété.

Une propriété liée génère un événement lorsque sa valeur change. D'autres objets peuvent s'enregistrer comme auditeurs pour un tel événement.

Ce mécanisme peut être mis en place grâce à un objet de la classe **PropertyChangeSupport** qui permet de simplifier la gestion de la liste des écouteurs et de les informer des changements de valeur d'une propriété. Cette classe définit les méthodes **addPropertyChangeListener ()** pour enregistrer un composant désirant être informé du changement de la valeur de la propriété et **removePropertyChangeListener ()** pour supprimer un composant de la liste.

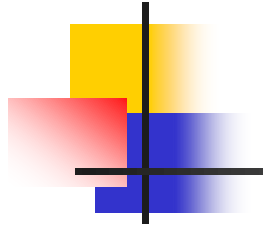
Les propriétés liées (Bound properties)



Un événement signifiant « *une de mes propriétés a changé* » est défini comme ceci :

```
public class java.beans.PropertyChangeEvent extends java.util.EventObject {  
    propertyChangeEvent (Object source, String propertyName, Object oldValue, Object  
    newValue);  
    public String getPropertyName ( );  
    public Object getOldValue ( );  
    public Object getNewValue ( );  
    public Object getPropagationId ( );  
    public void setPropagationId (Object id);  
}
```

Les propriétés liées (Bound properties)



Si cet événement est déclenché c'est qu'une propriété a été modifiée.

getPropertyName renvoie son nom ;

getOldValue et **getNewValue** son ancienne et sa nouvelle valeur, respectivement.

Notez que les valeurs (**oldValue**, **newValue**) sont des objets ; dans le cas de types primitifs, il faudra utiliser les « classes d'emballage » **Integer**, **Double**, etc.

L'identificateur de propagation (**propagationId**) est réservé pour un usage futur.

L'idée est la suivante : la source initiale de l'événement crée une valeur unique pour cet identificateur. Chaque objet qui réagit à l'événement en émettant un autre met dans le nouvel événement émis la même valeur d'identification. Cela doit permettre de repérer

Un objet qui a des propriétés liées est source d'événements **PropertyChangeEvent** ; il doit donc posséder les méthodes nécessaires à l'enregistrement des auditeurs.



```
public void addPropertyChangeListener (PropertyChangeListener l);  
public void removePropertyChangeListener (PropertyChangeListener l);
```

L'interface **PropertyChangeListener** est définie comme ceci :

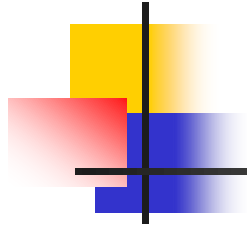
```
public interface PropertyChangeListener extends EventListener {  
    void propertyChange (PropertyChangeEvent evt);  
}
```

Donc les composants qui désirent être enregistrés doivent obligatoirement implémenter l'interface **PropertyChangeListener** et définir la méthode **propertyChange ()** déclarée par cette interface.

La méthode **propertyChange()** reçoit en paramètre un objet de type **PropertyChangeEvent** qui représente l'événement.

Cette méthode de tous les objets enregistrés est appelée. Le paramètre de type **PropertyChangeEvent** contient plusieurs informations (celles décrites plus haut) ie :

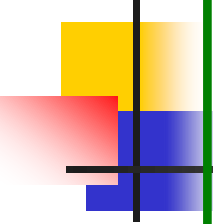
- *l'objet source* : le bean dont la valeur d'une propriété a changé
- *le nom de la propriété* sous forme de chaîne de caractères
- *l'ancienne valeur* sous forme d'un objet de type **Object**
- *la nouvelle valeur* sous forme d'un objet de type **Object**



Pour l'enregistrement des évènements, le bean peut hériter de la classe **PropertyChangeSupport** si possible car les méthodes *addPropertyChangeListener ()* et *removePropertyChangeListener ()* seront directement héritées.

Si ce n'est pas possible, il est obligatoire de définir les méthodes *addPropertyChangeListener ()* et *removePropertyChangeListener ()* dans le bean qui appelleront les méthodes correspondantes de l'objet **PropertyChangeSupport**.

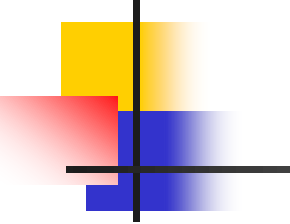
Voici un exemple:



```

import java.beans .*;
public class MonBean implements java.io.Serializable {
protected int valeur;
PropertyChangeSupport changeSupport;
public MonBean ( ){    valeur = 0;
                        changeSupport = new PropertyChangeSupport (this);
                        }
public synchronized void setValeur(int val) {
int oldValeur = valeur;
valeur = val;
/*La méthode firePropertyChange ( ) permet d'informer tous les composants
enregistrés du changement de la valeur de la propriété*/
changeSupport.firePropertyChange ("valeur",oldValeur,valeur);
}
public synchronized int getValeur ( ) {
return valeur;
}
public synchronized void addPropertyChangeListener (PropertyChangeListener
listener) { changeSupport.addPropertyChangeListener(listener);
            }
public synchronized void removePropertyChangeListener (PropertyChangeListener
listener) {
            changeSupport.removePropertyChangeListener (listener);
        } }

```



```
import java.beans.*; import java.util.*;
public class TestMonBean {
    public static void main (String[] args) {
        new TestMonBean ( );
    }

    public TestMonBean ( ) {
        MonBean monBean = new MonBean ( );
        monBean.addPropertyChangeListener ( new PropertyChangeListener ( ) {
            public void propertyChange (PropertyChangeEvent event) {
                System.out.println ("propertyChange : valeur = "+ event.getNewValue ( ));
            }
        } );
        System.out.println ("valeur = " + monBean.getValeur ( ));
        monBean.setValeur (10);
        System.out.println ("valeur = " + monBean.getValeur ( ));
    }
}
```

En sortie:

valeur = 0

propertyChange : valeur = 10

valeur = 10

Joseph NDONG UCAD DAKAR
SENEGAL



Remarque:

Supposons qu'on veuille que la propriété ne change que si sa valeur est supérieure à 100.
Et dans ce cas elle prend la valeur 333.

Voici comment on devra faire:

```
public class MonBean implements java.io.Serializable, PropertyChangeListener {  
.....  
public void propertyChange (PropertyChangeEvent evt) {  
if (evt.getPropertyName ( ) == "valeur") {  
double v = ((Integer) evt.getNewValue ( )).intValue ( );  
if (v > 100)  
valeur = 333 ;  
}  
.....  
}
```


Les propriétés liées avec contraintes (Constrained properties)

Ces propriétés permettent à un ou plusieurs composants de mettre un veto sur la *modification de la valeur de la propriété*.

Comme pour les propriétés liées, le bean doit gérer une liste de composants « écouteurs » qui souhaitent être informés d'un changement possible de la valeur de la propriété. Si un composant désire s'opposer à ce changement de valeur, il lève une exception pour en informer le bean.

Les écouteurs doivent implémenter *l'interface* **VetoableChangeListener** qui définit la méthode **vetoableChange ()**.

Avant le changement de la valeur, le bean appelle cette méthode **vetoableChange ()** de tous les écouteurs enregistrés. Elle possède en paramètre un objet de type **PropertyChangeEvent** qui contient : le bean, le nom de la propriété, l'ancienne valeur et la nouvelle valeur.

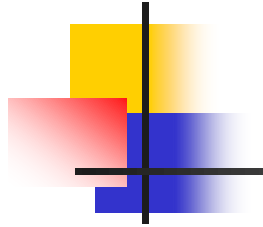
Si un objet écouteur s'oppose à la mise à jour de la valeur, il lève une exception de type **java.beans.PropertyVetoException**. Dans ce cas, le bean ne change pas la valeur de la propriété.

La classe **VetoableChangeSupport** permet de simplifier la gestion de la liste des écouteurs et de les informer du futur changement de valeur d'une propriété.

Son utilisation est similaire à celle de la classe **PropertyChangeSupport**.

ATTENTION: A cause de l'exception **PropertyVetoException** la méthode *set* est un peu différente pour une propriété contrainte :

```
public void set<nomPropriété> (<TypePropriété> valeur)  
throws java.beans.PropertyVetoException;
```



Un objet « demande la permission » de changer la valeur d'une propriété contrainte en notifiant un événement

PropertyChangeEvent à l'aide de la méthode **vetoableChange**.

L'objet doit donc posséder les deux méthodes

```
public void addVetoableChangeListener (VetoableChangeListener l);  
public void removeVetoableChangeListener (VetoableChangeListener l);
```

L'interface **VetoableChangeListener** est définie comme ceci :

```
public interface VetoableChangeListener extends EventListener {  
void vetoableChange (PropertyChangeEvent evt) throws PropertyVetoException;  
}
```

Les propriétés contraintes sont souvent aussi des propriétés liées. On notera que

- l'événement qui exprime la demande d'approbation,

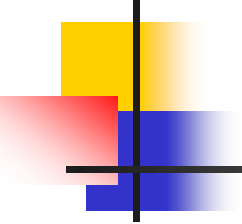
- vetoableChange** (PropertyChangeEvent evt), doit être émis *avant* la modification de la valeur de la propriété, l'événement qui exprime le changement,

- **propertyChange** (PropertyChangeEvent evt), doit être émis *après* la modification de la valeur de la propriété.

Exemple:

Ici on contrôle le changement (possibilité de modification) de la propriété valeur, si sa valeur dépasse 100, on pose un veto pour l'interdire.

```
import java.io.Serializable;
import java.beans.*;
public class MonBean implements Serializable {
    protected int oldValeur;
    protected int valeur;
    PropertyChangeSupport changeSupport;
    VetoableChangeSupport vetoableSupport;
    public MonBean ( ){
        valeur = 0;
        oldValeur = 0;
        changeSupport = new PropertyChangeSupport (this);
        vetoableSupport = new VetoableChangeSupport (this);
    }
}
```



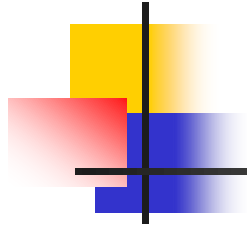
```

public synchronized void setValeur(int val) {
    oldValeur = valeur;
    valeur = val;

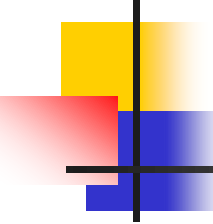
    try {
        vetoableSupport.fireVetoableChange("valeur",new Integer(oldValeur), new Integer(valeur));
    }
    catch (PropertyVetoException e) {
        System.out.println ("MonBean, un veto est emis : "+e.getMessage ( ));
        valeur = oldValeur;
    }
    if ( valeur != oldValeur ) {
        changeSupport.firePropertyChange ("valeur",oldValeur,valeur);
    }
}

public synchronized int getValeur ( ) { return valeur; }

```



```
public synchronized void addChangeListener (PropertyChangeListener listener) {  
    changeSupport.addPropertyChangeListener (listener);  
}  
public synchronized void removeChangeListener (PropertyChangeListener listener) {  
    changeSupport.removePropertyChangeListener (listener);  
}  
public synchronized void addVetoableChangeListener (VetoableChangeListener listener) {  
    vetoableSupport.addVetoableChangeListener (listener);  
}  
public synchronized void removeVetoableChangeListener (VetoableChangeListener listener) {  
    vetoableSupport.removeVetoableChangeListener (listener);  
}  
}
```



```

public class TestMonBean {
    public static void main (String [ ] args) {
        new TestMonBean ( );

        public TestMonBean ( ) {
            MonBean monBean = new MonBean ( );
            monBean.addPropertyChangeListener ( new PropertyChangeListener ( ) {
                public void propertyChange (PropertyChangeEvent event) {
                    System.out.println ("propertyChange : valeur = " + event.getNewValue ( ));
                }
            });
            monBean.addVetoableChangeListener ( new VetoableChangeListener ( ) {
                public void vetoableChange (PropertyChangeEvent event) throws
                PropertyVetoException {
                    System.out.println("vetoableChange : valeur = " + event.getNewValue ( ));
                    if ( ((Integer) event.getNewValue ( )).intValue ( ) > 100 )
                        throw new PropertyVetoException ("valeur superieur a 100",event);
                }
            });
            System.out.println ("valeur = " + monBean.getValeur ( ));
            monBean.setValeur (10);
            System.out.println ("valeur = " + monBean.getValeur ( ));
            monBean.setValeur (200);
            System.out.println ("valeur = " + monBean.getValeur ( ));
        }
    }
}

```

FIN

Joseph NDONG UCAD DAKAR