Java pour la programmation client/serveur

Julien Sopena

Julien.Sopena@lip6.fr

(basé sur un cours de **Gaël Thomas** et de **Lionel Seinturier**)

Université Pierre et Marie Curie

Master Informatique

M1 – Spécialité SAR

Java pour la programmation client/serveur

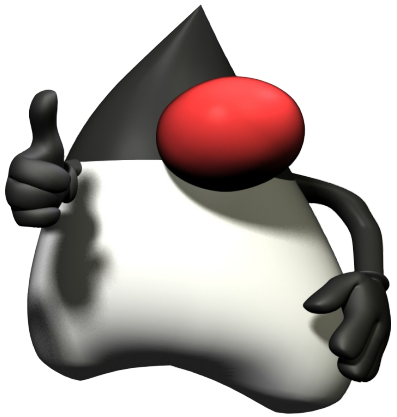
1. Rappels sur Java
   a. Java et les machines virtuelles
   b. Concepts de base
   c. Annexe : pense-bête
2. Entrées/sorties en Java
3. Programmation concurrente
   a. Introduction
   b. Les tâches en Java
   c. Moniteurs d'objets
   d. Envoi d'événements entre tâches Java
   e. Étude de deux patterns
4. Programmation réseau
   a. Introduction et rappels sur les sockets
   b. Sockets en mode flux
   c. Sockets en mode datagram
   d. Sockets en mode multicast

1. Rappels sur Java

a. Java et les machines virtuelles
b. Concepts de base
c. Java VS C++
d. Annexe : pense-bête

1.a. Java c'est quoi ?

- Un langage : Orienté objet fortement typé avec classes
- Un environnement d'exécution (JRE) :
  - ✓ Une machine virtuelle
  - ✓ Un ensemble de bibliothèques
- Un environnement de développement (JDK) :
  - ✓ Une machine
  - ✓ virtuelle et un ensemble d'outils
- Une mascotte : Duke

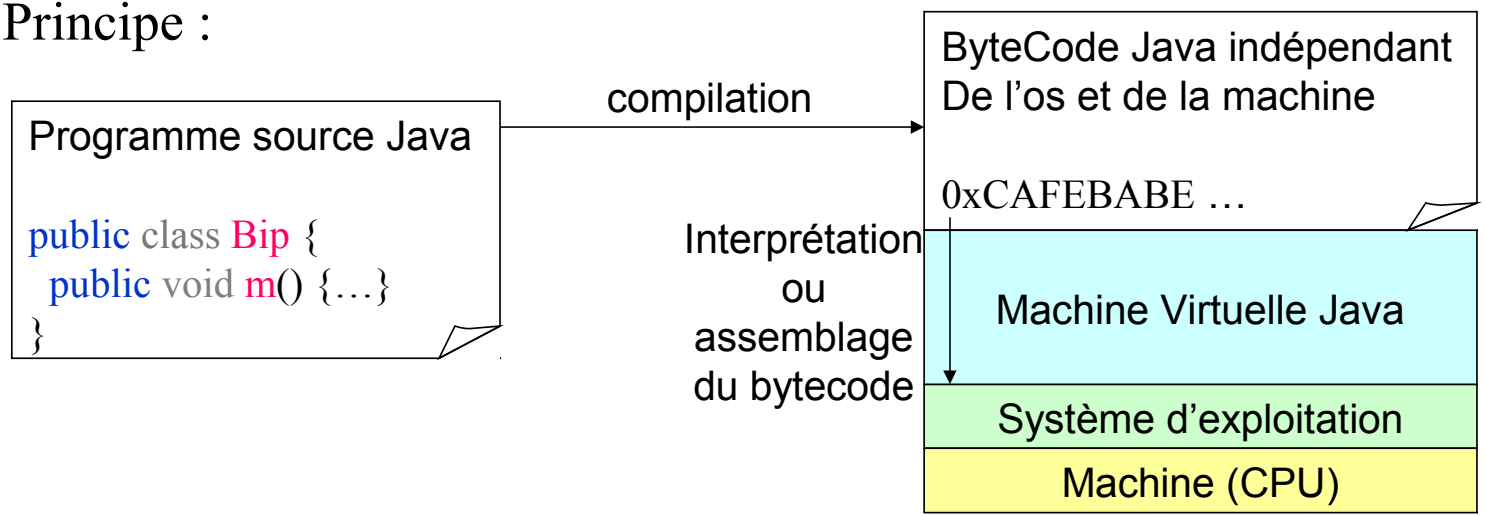


1.a. Java et les machines virtuelles

Objectifs de Java

- Masquer l'hétérogénéité des machines et des OS
  - ✓ Un binaire s'exécute à l'identique sur n'importe quelle machine…
- Fournir un langage de programmation orienté objet
- Fournir un ensemble de bibliothèques systèmes définies par des spécifications (J2ME, J2SE, J2EE)

Principe :



1.a. Java et les machines virtuelles

Historique

1991 : J. Gosling, B. Joy, A. Van Hoff proposent OAK

1993 : JDK 1.0, lien avec Web (applet)

1996 : Java Beans

1997 : JDK 1.1 Enterprise Java Beans, JavaCard, Java OS

1998 : Java 2 (JDK 1.2, 1.3, 1.4)

2004 : Java 5 (JDK 1.5)

2006 : Java 6

Grosse diff entre les 2 : bytecode compilé à la volée "JustInTime" des bouts de code (par fonction)

Optimisation "HOTSPOT" : Opti selon l'utilisation d'une fct (+ util° = + opti.) opti pour points chauds. HOTSPOT

Objets ds le TAS

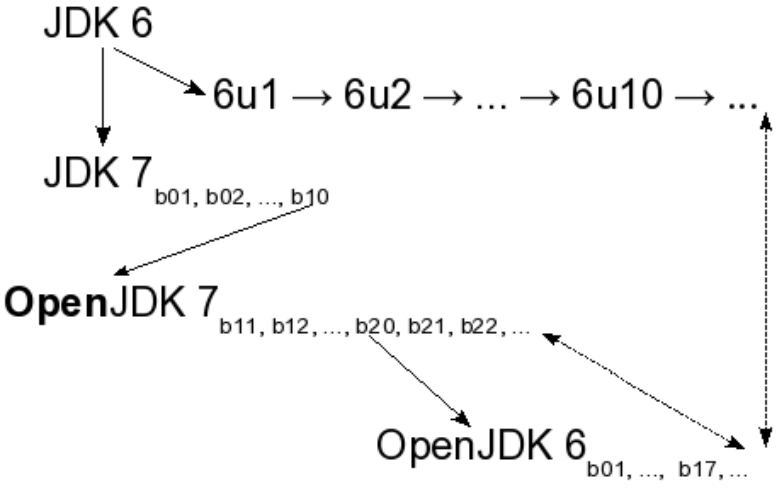
Nombreuses technologies pour les applications client/serveur

JSP, servlet, JDBC, JMS, JavaIDL, JavaMail, RMI, JCE, JAAS, JNDI, JTS, JTA, JMX, JPA…

Analyse d'échappement : est-ce que l'objet va être utilisé en dehors ? Non : alloc dans la pile

1.a. Java et les machines virtuelles

Attention, fin 2006 avec l'arrivée de la GPL tout se complique :



1.a. Java et les machines virtuelles

Java = 3 branches distinctes (compilo+VM+bibliothèques)

- JME : Java Micro Edition pour les systèmes embarqués
- JSE : Java Standard Edition pour le ordinateurs de bureau ("le" JDK)
- JEE : Java Enterprise Edition pour les serveurs (inclus J2SE)

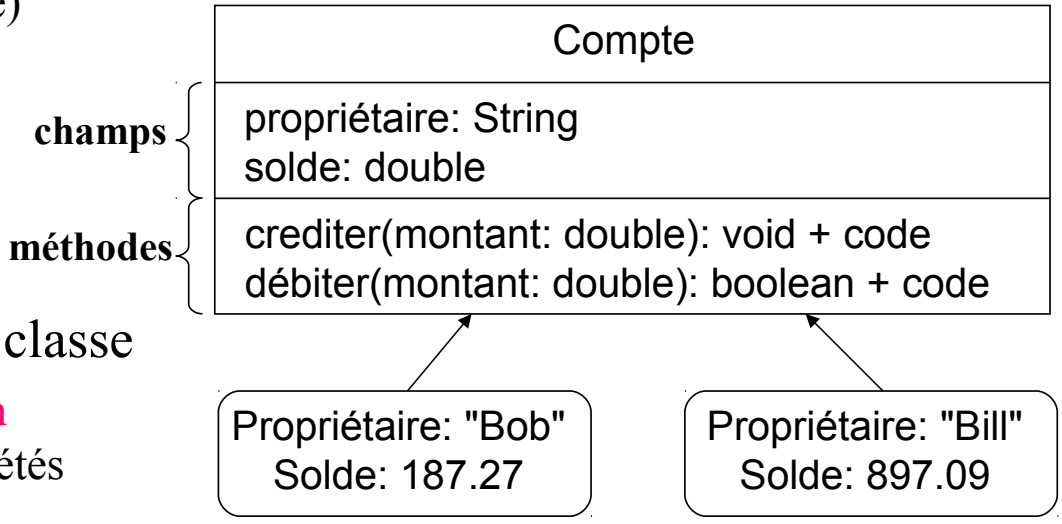
Nombreux outils

- Développement : Eclipse, JBuilder, NetBeans, emacs/jde…
- VM : Sun, Kaffe, Jikes RVM, SableVM, IBM, …
- Compilo : Sun, jikes, gcj, …
- Manipulation du bytecode : ASM, BCEL, Javassist, …

1.b. Concepts de base

Classe : élément de conception modélisant une entité du problème à résoudre, contient

- Les données de l'entité (variable)
- Les méthodes manipulant ces données (code)
- ✓ But : regrouper dans une même entité les données et leurs méthodes ⇒ seules les méthodes sont visibles (ce que fait une classe versus comment est construite une classe)



Objet : instance d'une classe

- Élément d'exécution possédant les propriétés de la classe

1.b. Concepts de base

2 types de champs et de méthodes

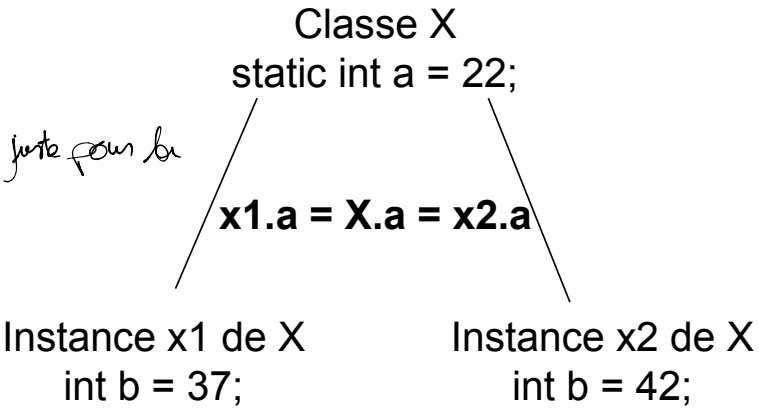
Champs et méthodes de classe :

- Mot clé : static ← membre de classe juste pour la classe.
- Partagés par toute les instances

⚠ return → Java : return void et pas int
main

Champs et méthodes d'instance :

- Mot clé : aucun (défaut)
- Liés à une instance



Remarque :

- Une méthode d'instance peut manipuler des champs de classe et d'instance
- Une méthode de classe ne peut manipuler que des champs de classe

1.b. Concepts de base

Instanciation et gestion de la mémoire

Instancier une classe X : appel **new X(arguments)**

- Allocation de l'espace mémoire de X
- Appel de la méthode spéciale X.X(arguments) appelée constructeur
- Retourne une référence vers l'instance de X
- Une référence n'est pas un pointeur (pas d'arithmétique)

Supprimer une instance de X : **automatique**

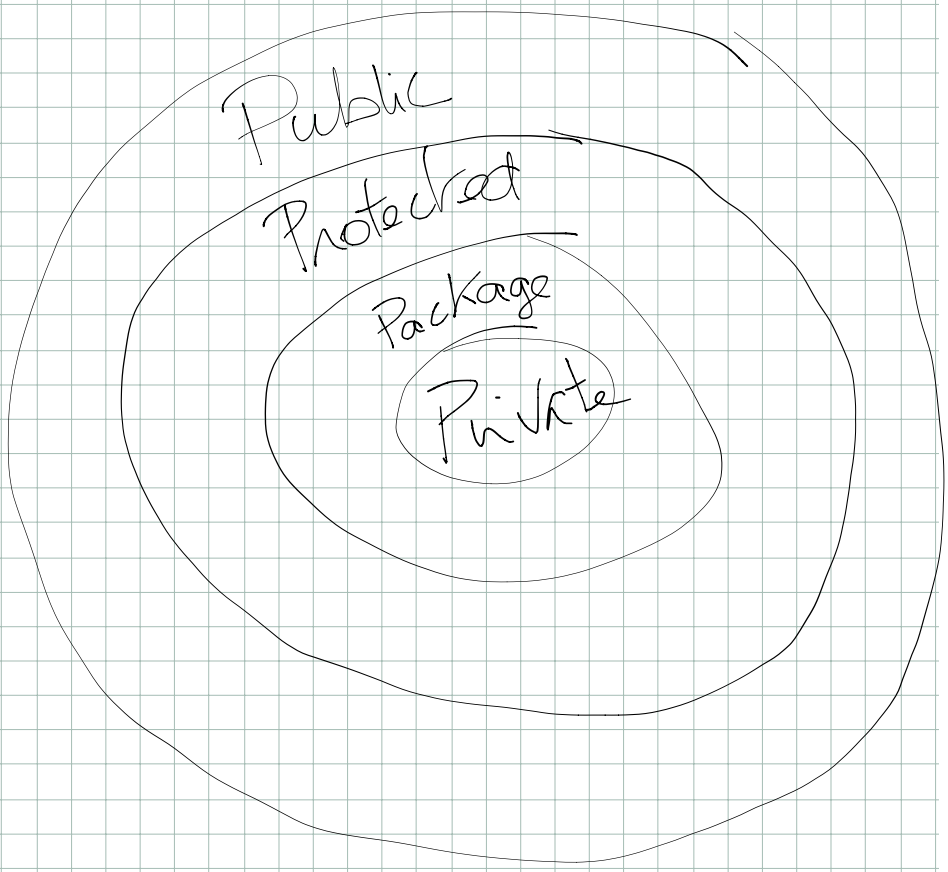
- Suppression automatique d'une référence dès qu'elle n'est plus référencée
- **Aucun contrôle sur l'instant de suppression!**
- Appel automatique de X.finalize() si définie

1.b. Concepts de base

Java par l'exemple (1/2)

Public / Protected / Package (—) / Private

```
public class Compte {
  private String proprietaire;  // private ⇒ invisible hors de la
   classe
  private double solde;

  public Compte(String proprietaire) {  // Constructeur, appelé lors
   de la création
    this.proprietaire = proprietaire; this.solde = 0; // this
   référence notre instance
  }

  public void crediter(double montant) { solde += montant; }

  public boolean debiter(double montant) { // public : visible en
   dehors de la classe
    if(solde >= montant) { solde -= montant;  return true;
    } else return false;
  }}
```



On peut être dans le package et si on ne spécifie pas package, c'est le répertoire qui le lance par défaut et lorsque l'on crée un paquet -> marche plus.

1.b. Concepts de base

Java par l'exemple (1/2)

Remarque : démarrage d'une application Java

Appel de `java Pgm` où `Pgm` est une classe Java

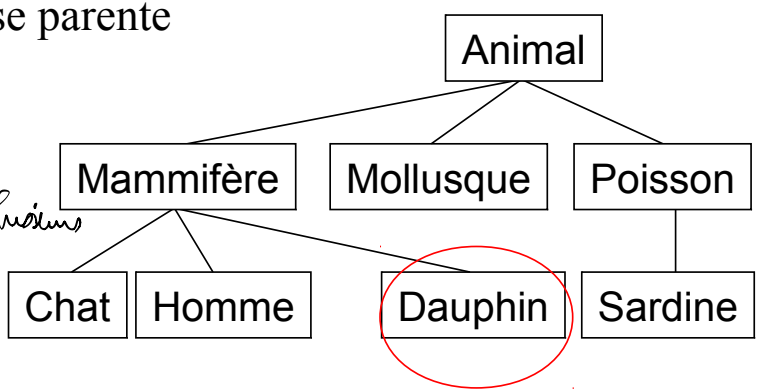
⇒Appel de la méthode `public static main(String args[])`

```
public class TestCompte {
  private static Compte bob;  // static ⇒ champs partagé entre toutes
   les instances
  private static Compte bill;

  public static void main(String args[]) { // type [] = tableau
    bob = new Compte("Bob");    // création du Compte de Bob
    bill = new Compte("Bill"); // création du compte de Bill
    bob.credite(187.12);
  }
}
```

1.b. Concepts de base

Héritage : relation de spécialisation d'une classe

- Une classe fille hérite d'une classe parente
  - Hérite des données et des méthodes de son parent
- Mot clé : **extends**



Pas d'héritage multiple en Java

On peut faire hériter des interfaces de plusieurs interfaces

Interface : définition abstraite d'une classe

- Ne possède que des méthodes
- Découple la définition d'une classe de son implantation
  - Lien avec la notion d'API
- ✓ Une interface peut être implantée (mot clé implements) par plusieurs classe
- ✓ Une classe peut posséder plusieurs interfaces

Interface : notion fondamentale pour séparer client et serveur

1.b. Concepts de base

Notion d'exception

- Exception = Instance d'une classe héritant de la classe **`Exception`**
- Peut être levée et attrapée

Erreur => Crash VM (mémoire)
Exception contrôlée : try-catch
Exception : RuntimeException (NullPointer)

Lever une exception : mot clé `throw`

- Indique un fonctionnement anormal du programme (erreur)
- Interrompt le fil d'exécution normal de l'application
- Reprise de l'exécution à
  - ✓ Un point où l'exception est attrapée si existe
  - ✓ Jamais sinon (interruption de l'application)
- Notion proche de l'envoie d'un signal à soit même (raise)

⚠ Division /0
↳ seulement pour les entiers

Attraper une exception : bloc clé `try { … } catch(Type t) { … }`

- Interception de la levée d'une exception et reprise de l'exécution
  - ✓ ayant la classe `Type`
  - ✓ Levée dans un des sous-appels contenu dans le bloc `try { … }`

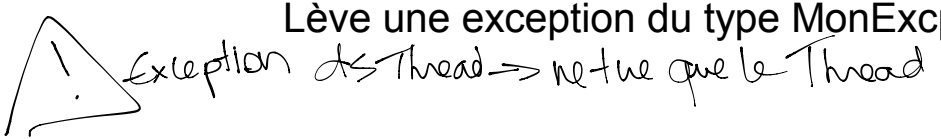
1.b. Concepts de base

Exception par l'exemple

```
public class MonExcp extends Exception { … }

public class Test {
  void f() throws MonExcp { … throw new MonException(); }

  void g() {
    try {
      f();
      System.out.println("Code jamais exécuté!!!");
    } catch(MonExcp e) {
      System.err.println("Exception interceptée : "  + e);
    }
  }
}
```

Indique que f est susceptible de lever une exception du type MonExcp

Lève une exception du type MonExcp

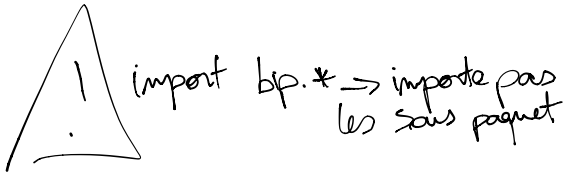


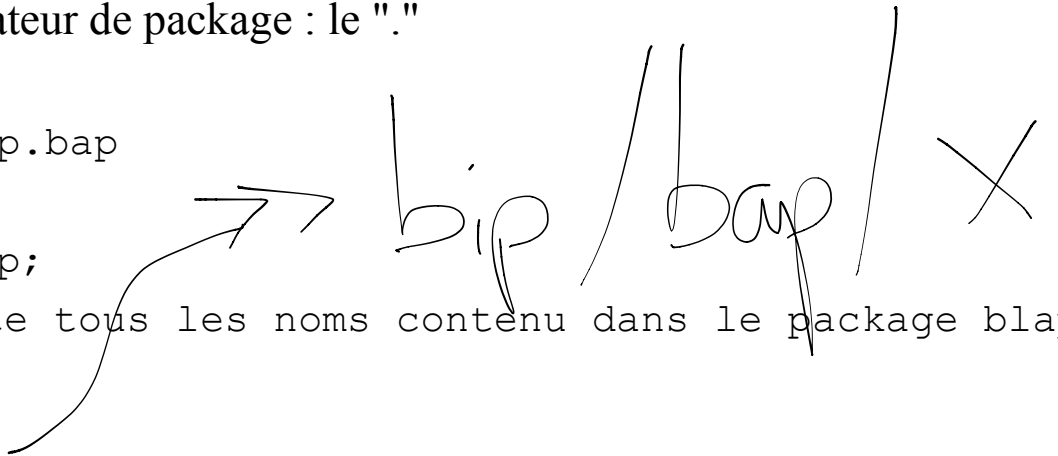
L'exécution reprend ici après la levée de l'exception

1.b. Concepts de base

Notion de package

- Espace de nommage regroupant des
  - ✓ Sous-packages (structure hiérarchique)
  - ✓ Classes ou des interfaces
- Séparateur de package : le "."






```
package bip.bap

import blap;
//⇒ importe tous les noms contenu dans le package blap

class X {
  void f() { System.out.println("blob.Y.y: " + blob.Y.y);
}
```

1.c. Java *versus* C++

- Filiation historique :
  - ✓ 1983 (AT&T Bell) : C++
  - ✓ 1991 (Sun Microsystems) : Java
- Java est **très proche** du langage C++ (et donc du langage C).
- Toutefois Java est **plus simple** que le langage C++ :
  - ✓ les points "critiques" du langage C++ ont été supprimés.

    ⇒ ceux qui sont à l'origine des principales erreurs
- Cela comprend :
  - ✓ Les pointeurs
  - ✓ Gestion de la mémoire
  - ✓ La surcharge d'opérateurs
  - ✓ L'héritage multiple

1.c. Java *versus* C++

- Avec Java tout est dynamique :
  - les instances d'une classe sont **instanciées dynamiquement**.
- La libération de mémoire est transparente pour l'utilisateur. Il n'est pas nécessaire de spécier de mécanisme de destruction.
- Elle est prise en charge un gestionnaire appelé :

  garbage collector ⇒ chargé de détecter les objets à détruire.

- **Surcoût :** perte en rapidité par rapport au C++.

+ **Gain de fiabilité :** pas de désallocation erronée.

1.c. Java *versus* C++

- Pour toujours plus de robustesse, Java inclus des vérifications :
  - ✓ type opérande,
  - ✓ taille de pile,
  - ✓ flot de données,
  - ✓ variable bien initialisé
  - ✓ …
- Certaines de ces vérications sont effectuées à la compilation du bytecode vers le langage natif du processeur, ralentissent l'exécution des classes Java.
- Mais les techniques de compilation à la volée "Just In Time (JIT)" ou **"Hotspot"** réduisent ce problème : elles permettent de ne traduire qu'une seule fois en code natif les instructions qui sont (souvent pour Hotspot) exécutées.

1.c. Java *versus* C++

Par rapport à C++, Java perd (un peu) en :

- **Efficacité**
- **Expressivité**

Mais gagne (beaucoup) en :

+ **Portabilité**
+ **Robustesse**

1.d. Pense-bête

- **CLASSPATH** : liste des répertoires dans lesquels la VM va chercher des .class?
- Comment **compiler** un fichier Toto.java?
  - javac Toto.java ⇒création de Toto.class
- Comment **exécuter** Toto.class?
  - java Toto
- Qu'est ce qu'un **package**?
  - Un ensemble de classes regroupées dans un espace de noms
  - package toto; public class Bip { … } ⇒la classe Bip est référencée par toto.Bip
- Comment **afficher** un message?
  - System.out.println("Hello, World!!!");
- Comment faire une **boucle**? Un **if** ?
  - for(int i=0; i<255; i++) { … }
  - if(cond) { si vrai } else { si faux }

1.d. Pense-bête

- Qu'est ce qu'une **exception**?
  - mécanisme de retour anticipé d'une méthode pour signaler un comportement anormal
- Comment traiter une **exception**?
  - try { m(); }
  - catch(MyException e) { System.out.println("Ce n'est pas normal"); e.printStackTrace(); }
- Comment lever une **exception**?
  - throw new MyException("Yeah!");
- Comment passer un **paramètre** à une application Java
  - java monpackage.Maclass arg0 arg1 arg2
- Comment positionner une **propriété** ?
  - java –Dfile.separator=\
- Comment consulter une **propriété** ?
  - String prop = System.getProperty("file.separator", "valeur par défaut");

⚠ Traitement par délégation des exceptions

2. Entrées/Sorties en Java

2. Entrées/sorties en Java

But : lire/écrire des données à partir d'un fichier, de la mémoire, du réseau…

Deux API d'entrées/sorties

- Entrées/sorties bloquantes (java.io.*) – étudiées dans ce cours
- Entrées/sorties non bloquantes (java.nio.*) – non abordées

Deux façon de gérer les entrées/sorties

- Par paquet : émetteurs et récepteurs communiquent directement par paquets de taille fixe – étudié avec UDP
- Par flux : émetteurs et récepteurs communiquent via un canal fifo, la taille des messages reçus et émis n'est pas la même – étudié dans ce chapitre

Deux mode d'entrée/sortie

- Mode binaire : écriture et lecture de données bruts – étudié dans ce cours
- Mode caractère : écriture et lecture de caractères – non abordé

2. Entrées/sorties en Java

InputStream :
lecture de données

int read();

Lit un octet du flux

Bloquant

Return -1 si fin du flux

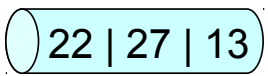
Bloquant, binaire, par flux

OutputStream :
écriture de données

void write(int b)

Écrit un octet dans le flux

```
InputStream is = …
int i = is.read();
while(i != -1) {
   … traitement …
   i = is.read();
}
is.close();
```



Flux

```
OutputStream os = …
os.write(22);
os.write(27);
os.write(13);
os.close();
```

2. Entrées/sorties en Java

FileInputStream :
lecture de données à partir d'un fichier (hérite de InputStream)

int read();

Lit un octet du fichier

Bloquant, binaire, par flux

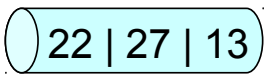
FileOutputStream :
écriture de données dans un fichier (hérite de OutputStream)

void write(int b)

Écrit un octet dans le fichier

```
InputStream is = new
FileInputStream(
  "/tmp/toto");
int i = is.read();
while(i != -1) {
  … traitement …
  i = is.read();
}
is.close();
```

```
OutputStream os = new
FileOutputStream(
  "/tmp/toto");
os.write(22);
os.write(27);
os.write(13);
os.close();
```

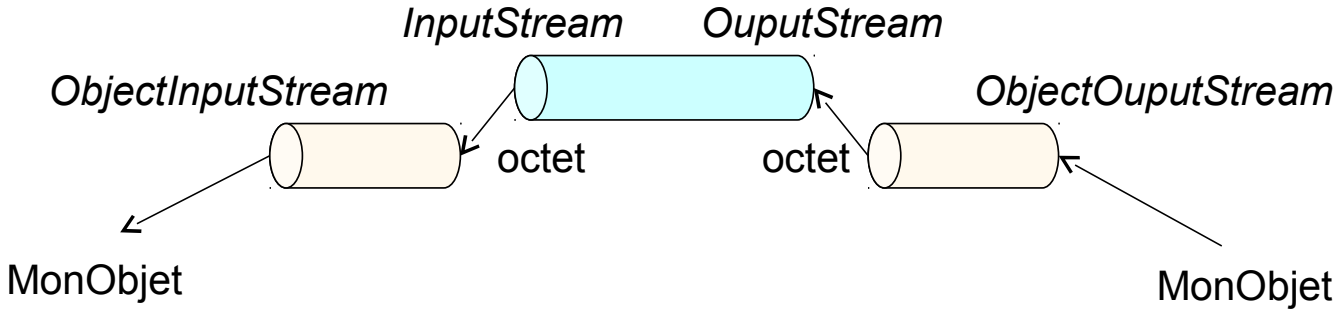


Flux

2. Entrées/sorties en Java

- **Sérialisation** : représentation sous forme binaire d'un objet Java
- Utilisé **pour échanger** des objets (envoi d'objets, persistance…)
- Ne marche qu'avec des Objets implantant l'interface Serializable

```
public class MonObject implements Serializable { … }
```



2. Entrées/sorties en Java : sérialisation

Envoi et réception d'objets Java : ObjectInputStream/ObjectOutputStream

ObjectInputStream :
lecture d'objets seralisables

Object readObject();

Lit un objet d'un flux

Repose sur un InputStream

Bloquant, binaire, par flux

ObjectInputStream :
écritur:we d'objets seralisables

void writeObject(Object o);

Écrit un objet dans un flux

Repose sur un OutputStream

```
InputStream is = new
ObjectInputStream(new
FileInputStream(
  "/tmp/toto"));

Object o = is.readObject();
```

```
OutputStream os = new
ObjectOutputStream(new
FileOutputStream(
  "/tmp/toto"));

os.writeObject(new MaClasse());
```

3. Programmation concurrente

a. Introduction
b. Les tâches en Java
c. Moniteurs d'objets
d. Envoi d'événements entre tâches Java
e. Étude de deux patterns

3.a. Introduction

Processus = ensemble d'instructions + état d'exécution

(pile, registres, pc, tas, descripteurs d'E/S, gestionnaires de signaux…)

Système multitâches : système capable d'exécuter en // plusieurs processus

- Optimisation des temps morts (idle, attente d'entrée/sortie)
- Utilisation multi-utilisateurs de la machine

Deux classes principales de processus

- Processus lourd (ou tâche ou processus) : ne partage pas son état

  Sauf des espaces mémoire partagés déclarés explicitement (IPC System V, shm_*)
- Processus léger (ou thread) : partage son tas, ses descripteurs et ses gestionnaires

3.a. Introduction

Intérêts des threads : communication inter-processus via la mémoire

- Programmation événementielle (IHM)
- Entrées/sorties non bloquantes
- Gestion de temporisateurs
- Servir plusieurs clients en parallèle

Défauts des threads (versus processus)

- Pas d'isolation : un thread corrompt la mémoire ⇒la mémoire est corrompue pour tous les threads
- Accès concurrent à l'état : 2 threads peuvent modifier en parallèle l'état

⇒ Plus difficile à programmer

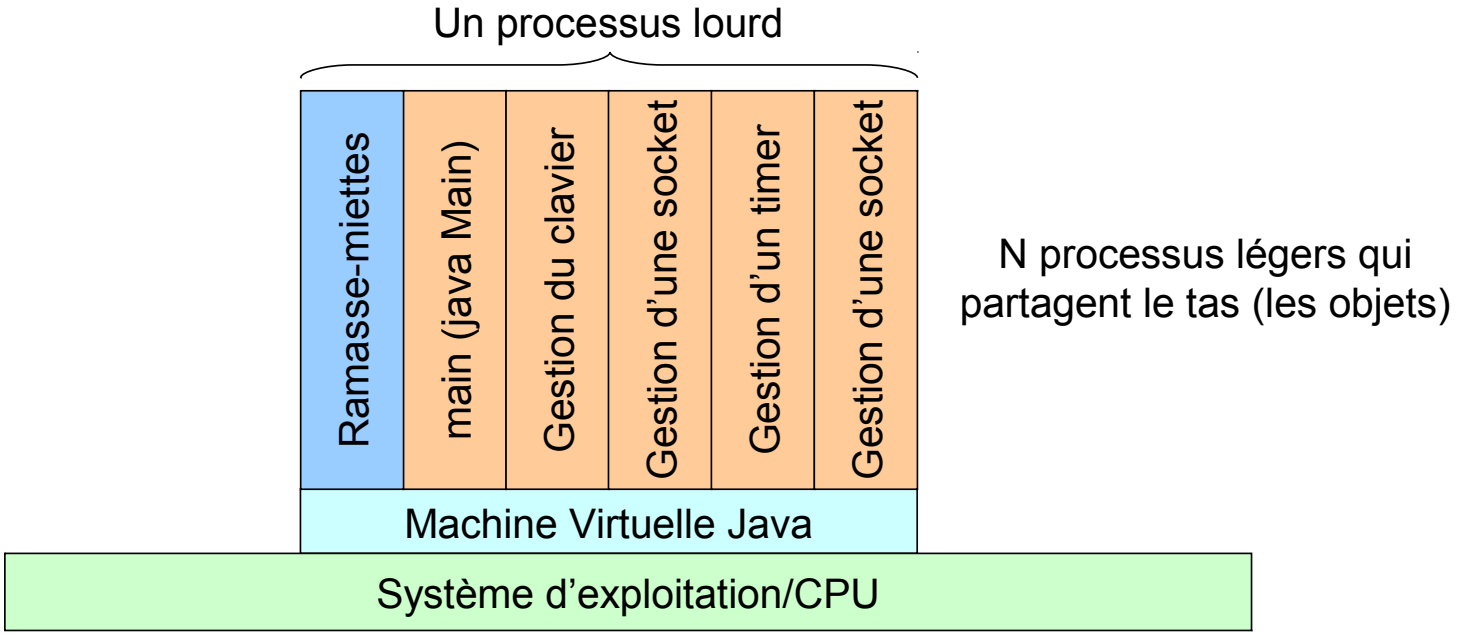
Systèmes actuels : approche mixte (processus lourds et légers)

3.b. Les Threads en Java

- **1 JVM = un processus lourd** (1 unique thread "main" au départ)
- Mécanisme de threads intégrés à la JVM

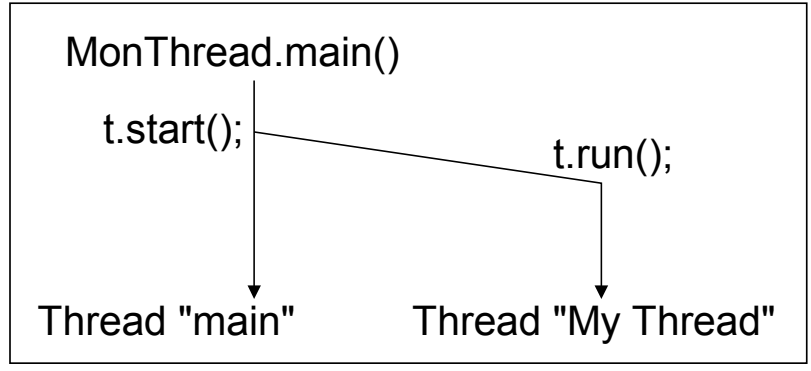
  (native threads : mappés sur processus noyau, green threads : en mode U)

  ⇒Tous les threads Java partagent la mémoire (les objets)



3.b. Les Threads en Java

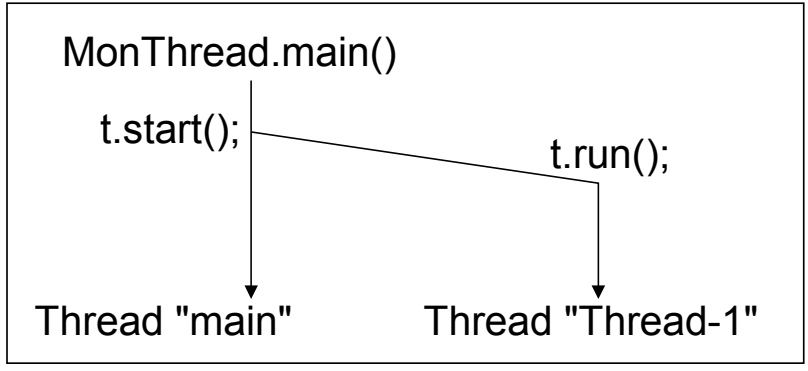
Processus léger Java = classe qui implante l'interface Runnable




```
class MonThread implements Runnable {
  public void run() {
    System.out.println("I'm a thread!");
  }

  public static void main(String args[]) {
    // Un thread possède optionnellement un nom symbolique
    Thread t = new Thread(new MonThread(), "My Thread");
    // MonThread.run() est démarré dans un nouveau thread après l'appel de start()
    t.start();
    System.out.println("Ce code s'exécute en // de run()");
  }
}
```

3.b. Les Threads en Java

La classe Thread implante aussi l'interface Runnable




```
class MonThread extends Thread {
  public void run() {
    System.out.println("I'm a thread!");
  }

  public static void main(String args[]) {
    // MonThread sera un Thread nommé automatiquement "Thread-1"
    Thread t = new MonThread();
    // MonThread.run() est démarré dans un nouveau thread après l'appel de start()
    t.start();
  }
}
```

3.b. Les Threads en Java

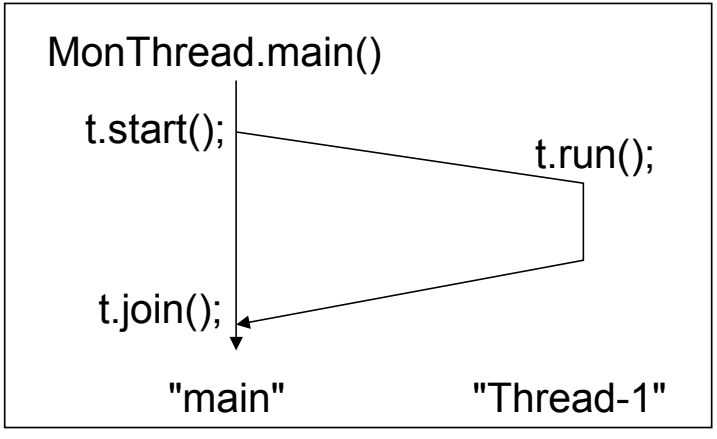
Remarques :

- Création d'autant de processus légers qui nécessaire
  Instances de la même classe ou d'autre classe
- **A une instance correspond un processus léger unique**
  Appel de start() une seule fois par instance de Thread
- Un thread meurt lorsque sa méthode run() se termine
- Pas de passage de paramètre directement à un Thread
  Passage d'argument via l'instance du Runnable

```
class MonThread extends Thread {
  private int value;
  MonThread(int value) { this.value = value; }
  public void run() { System.out.println("value is:" + value); }
  public static void main(String args[]) { (new MonThread(22)).start(); }
}
```

3.b. Les Threads en Java

Synchronisation sur la terminaison d'un Thread




```
class MonThread extends Thread {
  public void run() {
    System.out.println("I'm a thread!");
  }

  public static void main(String args[]) {
    Thread t = new MonThread();
    t.start();
    t.join(); // attend la fin de l'exécution du Thread
              // (i.e la fin de l'exécution de run())
  }
}
```

3.b. Les Threads en Java

Problème : terminaison et interruption des threads Java

- Il n'existe pas d'équivalent aux envois de signaux entre threads

- **t.interrupt()** : interrompt le thread t si
  - ✓ Il exécute wait(), sleep(), join()
  - ✓ Il effectue une entrée/sortie sur un InterruptibleChannel (interface java.nio), mais ⇒ fermeture du canal

- Si le thread effectue une entrée/sortie bloquante (java.net.*, java.io.*), il faut fermer le canal (flux, socket…)

⇒ Il n'existe pas de méthode pour interrompre un thread pendant qu'il effectue une entrée/sortie sans fermer le canal d'entrée/sortie (sauf si entrée/sortie avec temporisateur)

3.b. Les Threads en Java

Remarque : thread.stop() est obsolète

- Laisse les objets dans un état incohérent

```
public MonThread extends Thread {
  public void run() { hashmap.put(elmt, value); }

  public static void main(String args[]) {
    Thread t = new MonThread();
    t.start();
    t.stop();
  }
}
```

3.c. Moniteurs d'objets

Accès concurrent par l'exemple

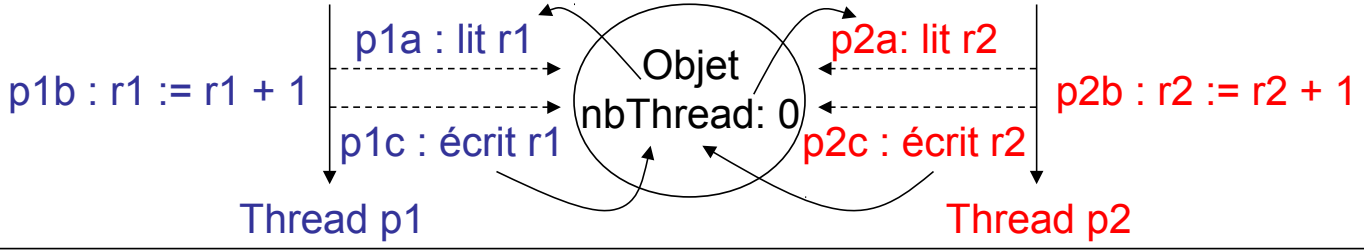
```
class Shared {       // Objet partagé entre les threads
  int nbThread = 0; // compte le nombre de thread qui  s'exécutent
  public void register() { nbThread++; } // incrémente ce nombre
  public String toString() { return "nbThread: " + nbThread; }
}

class MonThread extends Thread {
  static Shared shared = new Shared();// partagé par tous les threads

  public void run() {
    shared.register();
    System.out.println(shared);
  }

  public static void main(String args[]) {
    Thread t = new MonThread();
    t.start();
  }
}
```

3.c. Moniteurs d'objets

Plusieurs threads peuvent accéder à l'objet shared

- Problème classique d'accès concurrent à des variables
- Introduction de la notion de section critique

nbThread représente le nombre de thread, incrémenté par chaque Thread



Parmi les exécutions possibles (<nbThread, r1, r2>)

- p1a, p1b, p1c, p2a, p2b, p2c ⇒nbThread = 2 (gagné ☺)
  <0, 0,?>, <0,1,?>,<1,1,?>,<1,1,1>,<1,1,2>,<2,1,2>
- p1a, p2a, p2b, p2c, p1b, p1c ⇒nbThread = 1 (perdu ☹)
  <0,0,?>,<0,0,0>,<0,0,1>,<1,0,1>,<1,1,1>,<1,1,1>

3.c. Moniteurs d'objets

Définition : exclusion mutuelle (sémaphore à une case)

Seul un processus peut acquérir l'exclusion mutuelle

Les autres processus sont mis en attente

Exclusion mutuelle en Java

- Matérialisée par le mot clé *synchronized*
  - ✓ Acquisition d'un moniteur sur l'objet (⇔ sémaphore à une case)
  - ✓ Le moniteur est rendu à la fin du bloc synchronisé
- L'exclusion mutuelle s'effectue sur un objet, i.e l'exclusion concerne toutes les synchronisations sur cet objet

  Peut concerner des codes différents
- Méthode synchronisée ⇔ acquisition d'un moniteur sur un objet
  - ✓ Méthode d'instance synchronisée ⇔ synchronisation sur l'objet this
  - ✓ Méthode de classe synchronisée ⇔ synchronisation sur l'objet java.lang.Class

3.c. Moniteurs d'objets

Trois exemples totalement équivalents :

- synchronized public void register() { nbThread++; }
- public void register() { synchronized(**this**) { nbThread++; } }
- syncrhonized(shared) { shared.register(); … }

 Remarques : la synchronisation est à utiliser intelligemment

- Un appel à une méthode synchronisée coûte cher (3 fois plus cher)
- L'entrée dans un bloc synchronisé coûte aussi cher…
- Risque classique d'inter-bloquage

  P1 : synchronized(a) { synchronized(b) { … } }

  P2 : synchronized(b) { synchronized(a) { … } }

Autres remarques

La JVM garantie un accès atomique à tous les types primitifs sauf long et double

3.d. Envoi d'événements entre tâches Java

Moniteur insuffisant pour tous les types de synchronisations

- Synchronisation de threads sur un événement
- Exclusion de type lecteur/écrivain

Exemple : tous les thread attendent que tous les thread soient démarrés (attente d'un événement)

- Threads : attendent que le boolean partagé ready passe à true
  while(!shared.ready) { ??? dort(); ??? }
- Main : démarre les threads est passe le boolean ready à true
  shared.ready = true;
- Problème : comment éviter l'attente active (i.e comment endormir les thread)

3.d. envoi d'événements entre tâches Java

Quatre primitives de la classe Object pour éviter l'attente active

- **wait()** : attend un notify() ou un notifyAll();
- **wait(long ts)** : attend un notify()/notifyAll() ou ts milli-secondes
- **notify()** : réveille un de thread qui fait un wait()
- **notifyAll()** : réveille tous les threads qui font des wait()

Ces méthodes nécessitent un accès exclusif à l'objet

⇒à utiliser avec méthode ou bloc synchronized

- synchronized(obj) { obj.wait(); }
- synchronized(obj) { obj.notify(); }

Ces méthodes peuvent être interrompues (Thread.interrupt())

à entourer d'un try { .. } catch(InterruptedException e) { }

3.d. envoi d'événements entre tâches Java

Wait() :

- Mise en attente du thread
- Relâchement du moniteur (du verrou)
- Attente du notify() ou notifyAll()
- Attente de la ré-aquisition du moniteur (du verrou)
- Reprise de l'accès exclusif

Notify() :

- Si il existe des threads qui exécute wait(), réveilles en un unique
- Sinon, ne fait rien

A mettre en // des variables–conditions Posix (pthread_cond)

3.d. envoi d'événements entre tâches Java

```
class Shared {
  private boolean ready = false;

  public synchronized void waitReady() {
    while(!ready)
      try { wait(); } catch(InterruptedException e) {}
  }
  public synchronized void noteReady() {
    ready = true;
    notifyAll(); }
  }

public class Main extends Thread {
  private static Shared shared = new Shared();
  public void run() { shared.waitReady(); … }
  public static void main(String args[]) {
    … start threads …;
    shared.noteReady(); } }
```

3.d. envoi d'événements entre tâches Java

Schéma général de la synchronisation par événement

Thread qui attend un événement

```
synchronized(this) {
   while(!condition())
     try { wait(); } catch(InterruptedException e) {}
}
```

Thread qui génère l'événement

```
synchronized(this) {
  condition passe à true;
  notifyAll(); ou notify();
}
```

3.e. Étude de deux patterns

Problème : traitement en // de plusieurs clients par un serveur

- Création et destruction d'un thread par client (lent)
- Pas de limite au nombre de thread (serveur écroulé)

Solution : créer un ***pool*** de threads

- Les threads existent indépendamment des clients
- Les threads se voient assigner des clients au fur et à mesure

Deux approches :

- Pool statique : le nombre de threads du pool ne change jamais
- Pool dynamique : le nombre de threads du pool s'adapte dynamiquement au nombre de clients (tout en restant limité)

3.e. Étude de deux patterns

```
Thread  pool[];                  // contient les threads du pool + objet de synchro
Fifo    prochainClients;         // le nouveau client
boolean cont = true;             // drapeau pour la fin du programme

public void run() {
  Client client = null;

  while(cont) {
    synchronized(pool) {
      while(prochainClients.isEmpty())
          pool.wait();
      client = prochainClients.pop();
      prochainClient = null;
    }
    traitement(client);
  }
}
```

```
public void server() {
  while(cont) {
    Client client = attendClient();
    synchronized(pool) {
      prochainClients.push(client);
      pool.notify();
    }
  }
}
```

3.e. Étude de deux patterns

Problème : fonction d'écriture lente

Comment éviter de rester bloquer sur une écriture

Solution : utiliser un thread pour l'écriture

3.e. Étude de deux patterns

```
OutputStream          os = new …;  // le flux de sortie
LinkedList<Object>    pendings = new LinkedList<Object>(); // une liste chaînée
boolean               cont = true;    // indique la fin du programme

public void run() {
  Object next = null;

  while (cont) {
    synchronized (pendings) {
      while(pendings.size() == 0)
        pendings.wait();
     next = pendings.removeFirst();
    }
    os.writeObject (next);
  }
}
```

```
public void asyncWrite (Object o) {
  synchronized (pendings) {
    pendings.addLast(o);
    pendings.notify ();
  }
}
```

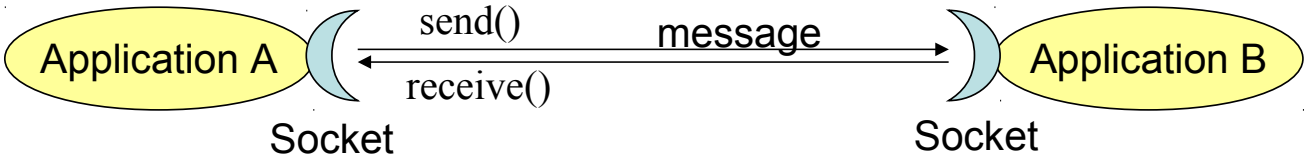
4. Programmation réseau

a. Introduction et rappels sur les sockets
b. Sockets en mode flux
c. Sockets en mode datagram
d. Sockets en mode multicast

4.a. Introduction et rappels sur les sockets

Socket = interface de programmation (API) avec les services du système d'exploitation pour exploiter les services de communication du système (local ou réseau)

Une socket est un demi-point de connexion d'une application
(≠ communication par message/MOM)



- Une socket est caractérisée par une adresse
- Plusieurs domaines de sockets existent :
  - Socket Unix (local) = un chemin dans le système de fichier
  - Socket Inet (réseau TCP, UDP ou IP) = adresse IP + port

4.a. Introduction et rappels sur les sockets

Couche 7	Applicative	Logiciels	NFS
Couche 6	Présentation	Représentation indépendante des données	XDR
Couche 5	Session	Établit et maintient des sessions	RPC
Couche 4	Transport	Liaison entre applications de bout en bout, fragmentation, éventuellement vérification	TCP, UDP, Multicast
Couche 3	Réseau	Adressage et routage entre machines	IP
Couche 2	Liaison	Encodage pour l'envoi, détection d'erreurs, synchronisation	Ethernet
Couche 1	Physique	Le support de transmission lui-même	

- Sockets en Java : uniquement orientée transport (couche 4)
- Deux API pour les sockets
  - java.net : API bloquante (étudié ici)
  - java.nio.channels (> 1.4) : API non bloquante (non étudiée dans ce cours)

4.a. Introduction et rappels sur les sockets

Mode connecté : la communication entre un client et un serveur est précédée d'une connexion et suivi d'une fermeture

- Facilite la gestion d'état
- Meilleurs contrôle des arrivées/départs de clients
- Uniquement communication unicast
- Plus lent au démarrage

Mode non connecté : les messages sont envoyés librement

- Plus facile à mettre en œuvre
- Plus rapide au démarrage

- Il ne faut pas confondre connexion au niveau transport et au niveau applicatif!
  - HTTP est un protocole non connecté alors que TCP l'ai
  - FTP est un protocole connecté et TCP aussi
  - RPC est un protocole non connecté et UDP non plus

4.a. Introduction et rappels sur les sockets

Liaison par **flux** : Socket/ServerSocket (TCP)

- Connecté : protocole de prise de connexion (lent)
  ⇒communication uniquement point à point
- Sans perte : un message arrive au moins un fois
- Sans duplication : un message arrive au plus une fois
- Avec fragmentation : les messages sont coupés
- Ordre respecté
- ✓ Communication de type téléphone

Liaison par **datagram** : DatagramSocket/DatagramPacket (UDP)

- Non connecté : pas de protocole de connexion (plus rapide)
- Avec perte : l'émetteur n'est pas assuré de la délivrance
- Avec duplication : un message peut arriver plus d'une fois
- Sans fragmentation : les messages envoyés ne sont jamais coupés
  ⇒soit un message arrive entièrement, soit il n'arrive pas
- Ordre non respecté
- ✓ Communication de type courrier

4.a. Introduction et rappels sur les sockets

Une socket est identifiée par

- Une adresse IP : une des adresses de la machine (ou toutes)
- Un port : attribué automatiquement ou choisi par le programme

Adresse de Socket = Adresse IP + port

Une socket communique avec une autre socket via à son adresse

- Flux : une socket se connecte à une autre socket via son adresse de socket
- Datagram : une socket envoie/reçoit des données à/d'une autre socket identifiée par son adresse de socket

4.a. Introduction et rappels sur les sockets

Adressage

- Une adresse IP : identifie une carte réseau d'une machine
  (par exp : 195.83.118.1)
  ⇒une machine peut posséder plusieurs adresses (penser aux routeurs)
- Un port : identifie l'application (par exp 21/ftpd)
  Seul l'administrateur peut ouvrir des ports < 1024

Nom symbolique (Domain Name Server)

- Associe une adresse IP à un nom symbolique
  (par exp ftp.lip6.fr => 195.83.118.1)
- Une adresse peut être associée à plusieurs noms
  (par exp nephtys.lip6.fr ⇒ftp.lip6.fr ⇒195.83.118.1)

Classes d'adresses : A (1-126), B (128-191), C (192-223),
D/Multicast (224-239), Locale (127)

4.a. Introduction et rappels sur les sockets

java.net.InetAddress : objet représentant une adresse IP

static InetAddress InetAddress.getByAddress(byte ip[])	construit un objet d'adresse ip
static InetAddress InetAddress.getByName(String name)	renvoie l'Adresse IP de name
static InetAddress InetAddress.getLocalHost()	renvoie notre adresse
String InetAddress.getHostName()	renvoie le nom symbolique
byte[] InetAddress.getHostAddr()	renvoie l'adresse IP

```
public class Main {
  public static void main(String args[]) {
    byte ip[] = {195, 83, 118, 1 };
    InetAddress addr0 = InetAddress.getByAddress(ip);
    InetAddress addr1 = InetAddress.getByName("ftp.lip6.fr");
    …
  }
}
```

4.a. Introduction et rappels sur les sockets

java.net.SocketAddress : objet représentant une adresse de Socket sans protocole attaché

java.net.InetSocketAddress : objet représentant une adresse IP + port

InetSocketAddress(InetAddress addr, int port);	Construit une adresse de Socket
InetSocketAddress(String name, int port);	Construit une adresse de Socket
InetAddress InetSocketAddress.getAddress()	Renvoie l'adresse IP
int InetSocketAddress.getPort();	Renvoie le port
String InetSocketAddress.getHostName();	Renvoie le nom symbolique de l'IP

```
public class Main {
  public static void main(String args[]) {
    byte ip[] = {195, 83, 118, 1 };
    InetAddress addr = InetAddress.getByAddress(ip);
    InetSocketAddress saddr0 = new InetSocketAddress(addr, 21);
    InetSocketAddress saddr1 = new InetSocketAddress("ftp.lip6.fr", 21);
    … } }
```

4.a. Introduction et rappels sur les sockets

Problème : plusieurs clients **colocalisés** doivent se connecter à un serveur unique

- Temps d'ouverture/fermeture de connexion long
- Tous les clients ne sont pas forcement connectés à chaque instant
- Apparition/disparition de clients

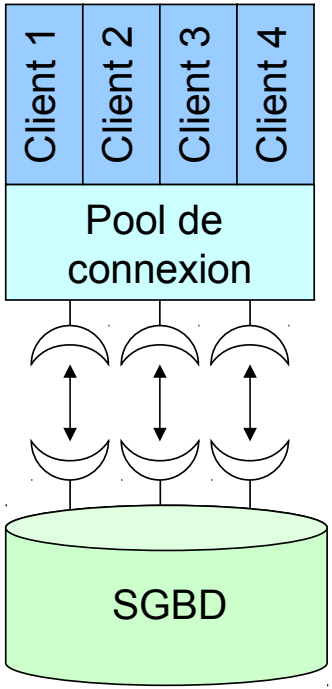
Solution : mutualiser les connexions

- Pool de connexions ouvertes en permanence
- Les clients (ré)utilisent les connexions ouvertes

Exemple : pool de connexions à un SGBD

⇒ Politique de gestion de ressources partagées

Problèmes classiques de réservation de ressources, d'interblocages…



4.a. Introduction et rappels sur les sockets

Problème :

- Passage de firewalls
- Optimisation du nombre de connexions

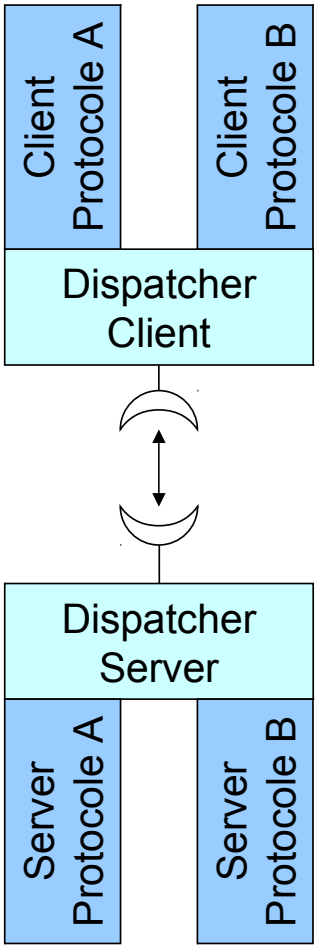
Solution : Multiplexer une connexion

- Plusieurs protocoles transitent par la même socket

⇒ Distinguer les flux de données

- Encadrer les protocoles par des méta-données
- Acheminer le message vers la bonne application

Peut être couplé avec un pool de connexions



4.a. Introduction et rappels sur les sockets

Représentation données

- Pas le même codage entre serveur et client
  big endian (powerpc) versus little indian (pentium)
- Pas la même façon de stocker les données
  processeur 32 bits (pentium) versus processeur 64 bits (xeon)

Deux solutions pour communiquer

- Prévoir toutes les conversions possibles (n^2 convertisseurs)
- Utiliser un format pivot (2n convertisseurs)

Nombreux formats pivots :

Sun XDR, sérialisation Java, Corba CDR…

4.b. Sockets en mode flux

Socket en mode flux de Java

- Repose sur TCP
- Indépendant de TCP

Propriétés

- Taille des messages quelconques
- Envoi en général bufferisé (i.e. à un envoi OS correspond plusieurs écritures Java)
- Pas de perte de messages, pas de duplication
- Les messages arrivent dans l'ordre d'émission
- Contrôle de flux (i.e. bloque l'émetteur si le récepteur est trop lent)
- Pas de reprise sur panne
  Trop de perte ou réseau saturé ⇒connexion perdue

Nombreuses utilisations : HTTP, FTP, Telnet, SMTP, POP…

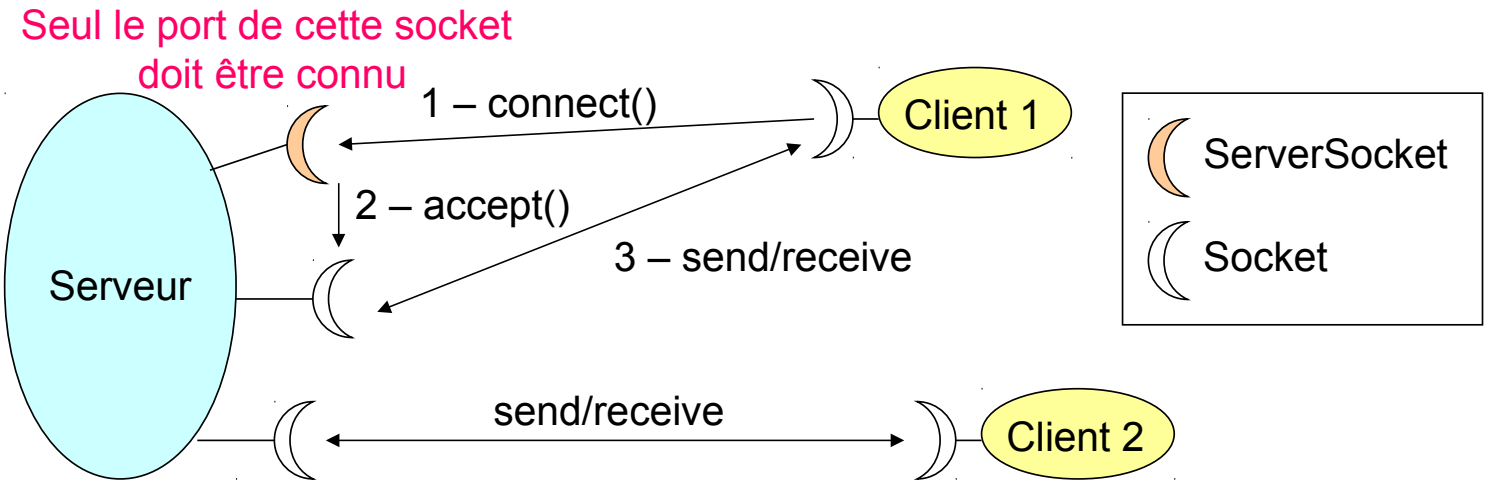
4.b. Sockets en mode flux

Nécessite une phase de connexion

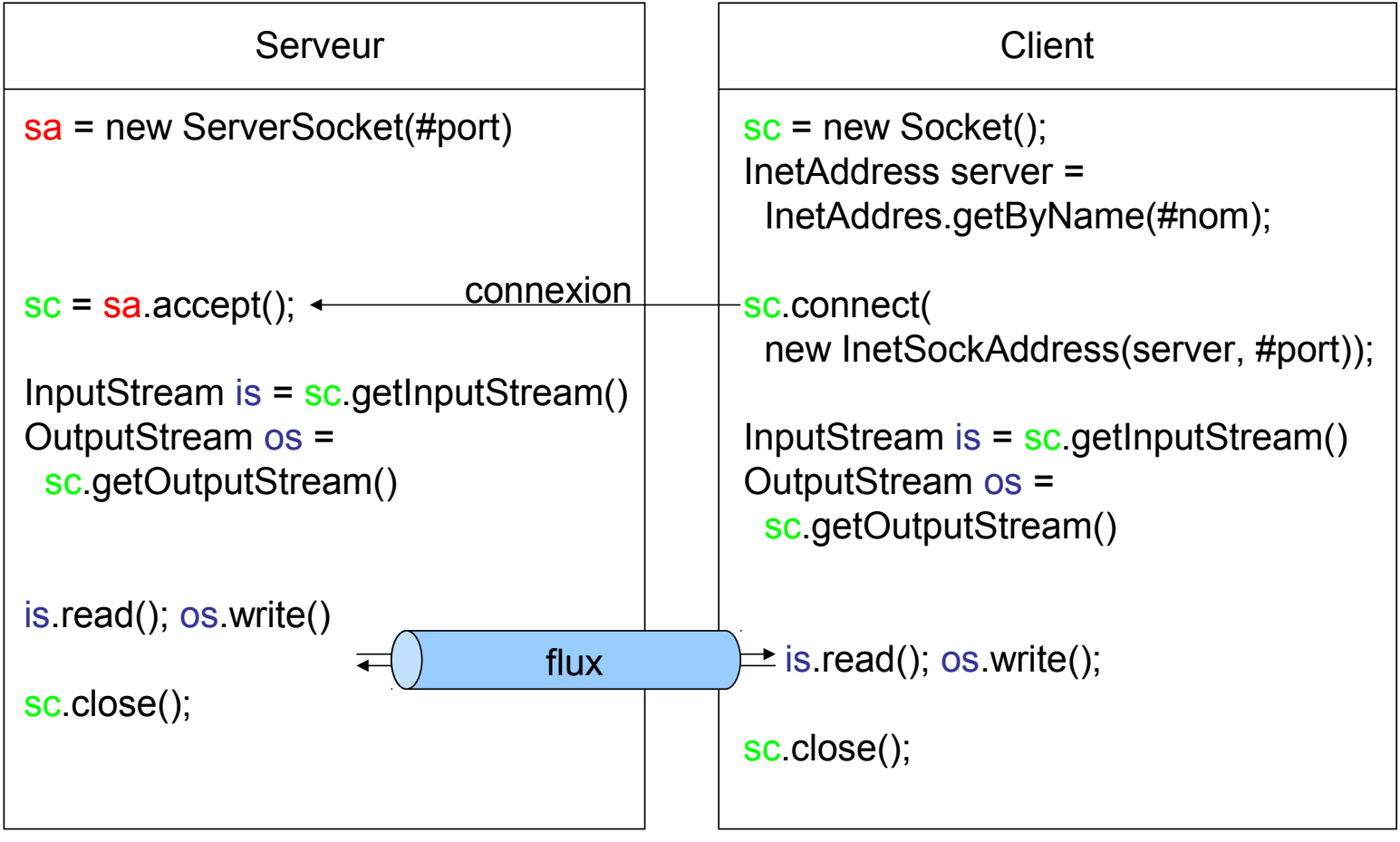
- Serveur : attend des connexion ⇒une socket de connexion (ServerSocket)
- Client : se connecte au serveur (Socket)

Le serveur doit maintenir des connexions avec plusieurs clients

⇒ Une socket de communication par client (Socket)



4.b. Sockets en mode flux



4.b. Sockets en mode flux

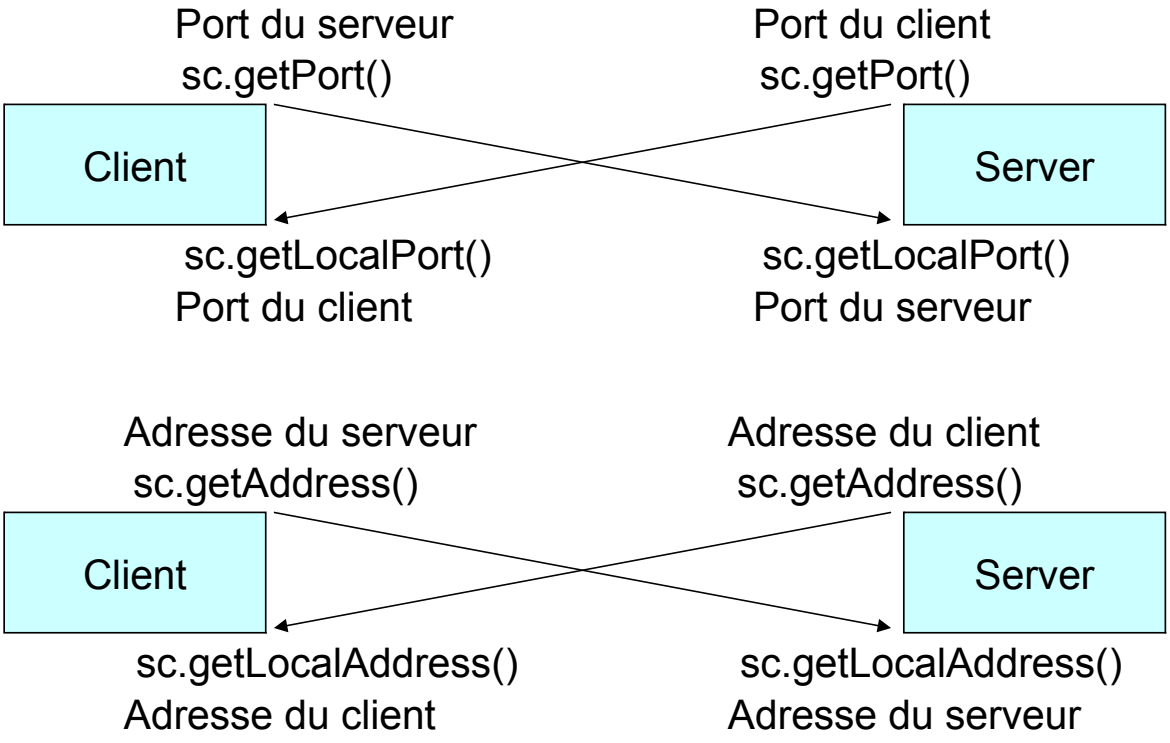
Remarques

- La socket du serveur possède le port #port qui identifie le serveur
- La socket de communication du serveur possède un port attribuée automatiquement par Java lors de l'accept()
- La socket de communication du client possède un port attribuée automatiquement par Java lors du connect()

- Il est possible de fixer le ports de la socket du client par
  sc.bind(new SocketAddress(InetAddress.getLocalHost(), #port));
- Il est possible de fixer le port de la socket de connexion du serveur après sa création par
  ServerSocket sa = new ServerSocket();
  sa.bind(**new** SocketAddress(InetAddress.getLocalHost(), #port));

4.b. Sockets en mode flux

Retrouver les adresses IP et les ports



4.b. Sockets en mode flux

Remarque : les flux associés aux sockets peuvent être encapsulés dans n'importe quels autres flux

```
ObjectInputStream is =
    new ObjectInputStream(
      new GZipInputStream(sc.getInputStream()));

ObjectOutputStream os =
    new ObjectOutputStream(
      new GZipOutputStream(sc.getOuputStream()));
```

Émission : os.writeObject("Hello, World!!!");

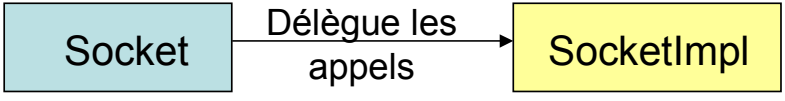
Réception : System.out.println(is.readObject());

4.b. Sockets en mode flux

Problème : définir des sockets personnalisées

(Passer des firewalls, utiliser un autre protocole que TCP)

Solution : Personnalisation des sockets



- Principe : une Socket délègue ses méthodes à un objet SocketImpl
  (accept(), connect(), getInputStream(), getOutputStream()…)
- Personnaliser les Sockets ⇔
  - ✓ Définir une sous-classe de SocketImpl
  - ✓ Définir une classe qui implante SocketImplFactory
    ```
    interface SocketImplFactory {
      SocketImpl createSocketImpl();
    }
    ```
  - ✓ Associer notre SocketImplFactory à la factory par défaut avec
    `static void setSocketFactory(SocketImplFactory fac);`

⇒ Toute nouvelle Socket créée utilisera notre SocketImpl

4.c. Sockets en mode datagram

Socket en mode datagram de Java

- Repose sur UDP
- Indépendant de UDP

Propriétés

- Taille des messages fixe et limitée (64ko)
- envoi non bufferisé
- Possibilité de perte de messages, Duplication
- Les messages n'arrivent pas forcement dans l'ordre d'émission
- Aucun contrôle de flux
- Pas de détection de panne (même pas assuré que les messages arrivent)
- **Faible latence** (car aucun contrôle de flux, pas de connexion)

Nombreuses utilisations :

- DNS, TFTP, RIP…
- Base pour la construction de IP Multicast

4.c. Sockets en mode datagram

DatagramSocket : Socket orientée Datagram

- Liée à un port
  - ✓ Assignation explicite new DatagramSocket(#port)
    ou socket.bind(SocketAddress saddr);
  - ✓ Assignation automatique lors de la première entrée/sortie
- Communique uniquement via des DatagramPacket (pas de flux!)

DatagramPacket : représente un message

- En réception : DatagramPacket(byte buf[], int offset, int length);
- En émission : DatagramPacket(byte buf[], int offset, int length, InetSocketAddress saddr);

 Attention : si la taille du buffer de réception est trop petite, la fin du message est perdu!

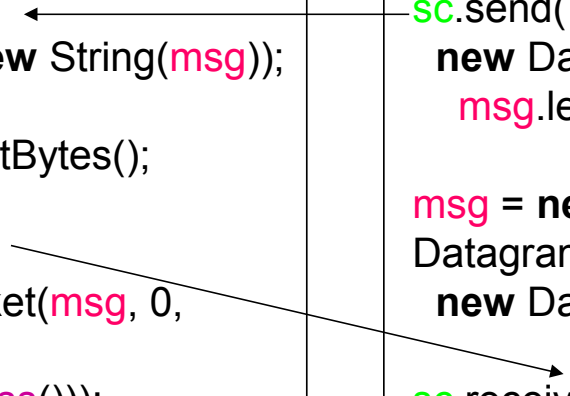
4.c. Sockets en mode datagram

Serveur

```
sc = new DatagramSocket(#port);

byte msg[] = new byte[256];
DatagramPacket in =
  new DatagramPacket(msg, 256);

sc.receive(in);
System.out.println(new String(msg));

msg = "Bien recu".getBytes();

sc.send(
  new DatagramPacket(msg, 0,
    msg.length,
    in.getSocketAddress()));
```

Client

```
sc = new DatagramSocket();
InetAddress server =
  InetAddres.getByName(#nom);

byte msg[] =
  "Hello, World!!!".getBytes();
sc.send(
  new DatagramPacket(msg, 0
    msg.length, server, #port);

msg = new byte[256];
DatagramPacket in =
  new DatagramPacket(msg, 256);

sc.receive(in);
System.out.println(new String(msg));
```



4.c. Sockets en mode datagram

Remarques sur les ports

- Le client a besoin de connaître l'adresse IP et le port du serveur
- La socket du client se voit assigner un port lors de l'envoi

Remarques sur la sérialisation

- Une DatagramSocket ne possède pas de flux (car ce n'est pas un flux!!!)
- Envoi uniquement de tableaux de bytes
- Réception uniquement de tableaux de bytes

La sérialisation est tout de même possible en utilisant

- ByteArrayInputStream : flux d'entrée qui lit à partir d'un tableau de bytes
- ByteArrayOutputStream : flux de sortie qui écrit dans un tableau de bytes
- Les messages restent limités en taille et le récepteur doit prévoir à priori un tampon suffisamment grand

4.c. Sockets en mode datagram

```
DatagramPacket serialize(Object o) {
  ByteArrayOutputStream bos = new ByteArrayOutputStream();
  ObjectOutputStream os = new ObjectOutputStream(bos);
  os.writeObject(o);
  byte msg[] = bos.toByteArray();
  return new DatagramPacket(msg, msg.length);
}

Object deserialize(DatagramPacket packet) {
  ObjectInputStream is =
    new ObjectInputStream(
      new ByteArrayInputStream(packet.getData(),
                               packet.getOffset(),
                               packet.getLength());
  return is.readObject();
}
```

```
Object receive(DatagramSocket s) {
  byte buf[] = new byte[????];
  DatagramPacket packet =
    new DatagramPacket(buf,
      buf.length);
  s.receive(packet);
  return deserialize(packet);
}
```

Comment prévoir la taille des messages?

4.c. Sockets en mode datagram

Problème : définir des sockets personnalisées

(Passer des firewalls, utiliser un autre protocole que UDP, crypter les communication…)

Solution : Personnalisation des sockets

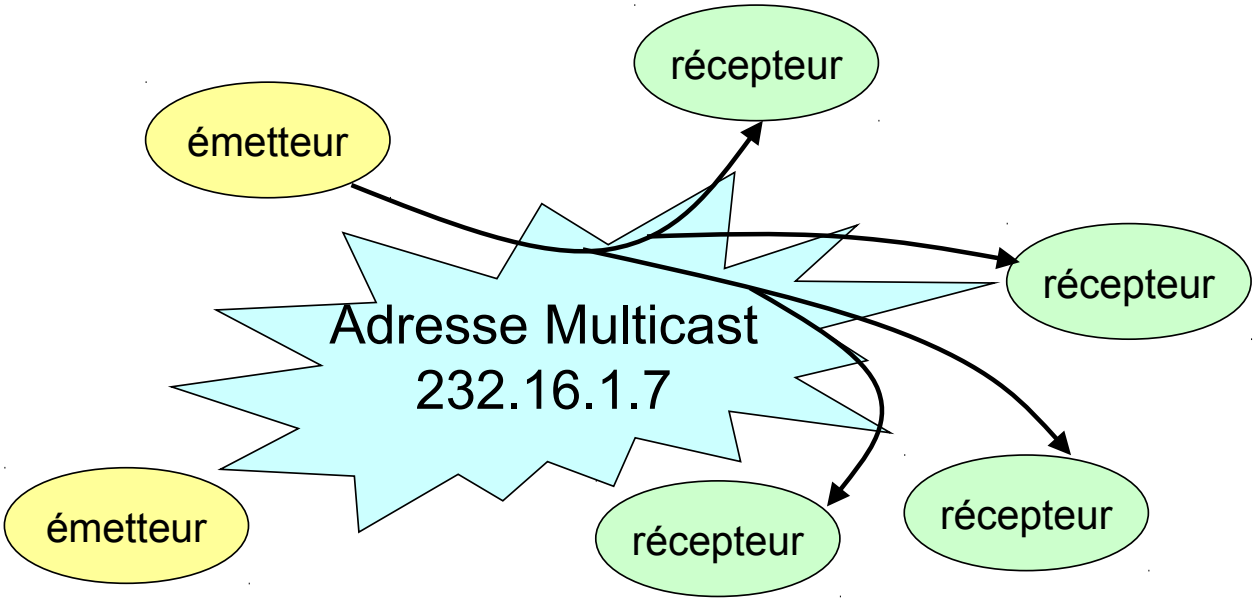
Même principe qu'avec les sockets en mode flux

- Définir une nouvelle implantation de DatagramSocket via DatagramSocketImpl
- Définir une nouvelle usine à DatagramSocket via DatagramSocketImplFactory
- Enregistrer l'usine via DatagramSocket.setDatagramSocketImplFactory(…)

4.d. Sockets en mode multicast

Multicast = diffusion de groupe

- Récepteur : s'abonne à une adresse IP de classe D
  - Adresse IP comprise entre 224.0.0.0 et 239.255.255.255
  - Certaines adresses sont déjà réservées
    - (voir http://www.iana.org/assignments/multicast-addresses)
- Émetteurs : émettent à destination de cette adresse IP



4.d. Sockets en mode multicast

Socket en mode multicast de Java

- Repose sur IP Multicast, lui-même basé sur UDP
- Indépendant de UDP

Propriétés : même propriétés qu'UDP

- Taille des messages fixe et limitée (64ko)
- envoi non bufferisé
- Possibilité de perte de messages ou de duplication pour certains récepteurs
- Les messages n'arrivent pas forcement dans l'ordre d'émission, pas forcement dans le même ordre chez tous les récepteurs
- Aucun contrôle de flux

Encore peu d'utilisations :

- Université
- Certaines webradio, certains fournisseurs d'accès pour de la diffusion vidéo
- La plupart des routeurs jettent les packets multicast!

4.d. Sockets en mode multicast

Groupe multicast = ensemble de récepteurs sur une adresse multicast

Émetteur :

- Émet sur une adresse de classe D + port
- Pas d'abonnement nécessaire
- Émission à tout instant

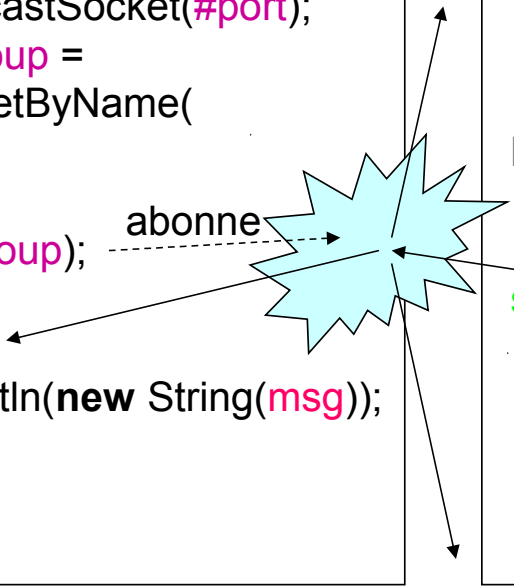
Récepteur :

- S'abonne à une adresse de classe D (⇒abonnement de la machine)
  MulticastSocket.joinGroup(InetAddress group);
- Écoute sur un port donnée (⇒abonnement de l'application)
- Se désabonne de la classe D avant de quitter
  MulticasrSocket.leaveGroup(InetAddress group);
- Peut rejoindre et quitter le groupe multicast à tout instant

4.d. Sockets en mode multicast

Récepteur

```
byte msg[] = new byte[256];
DatagramPacket in =
  new DatagramPacket(msg, 256);

sc = new MulticastSocket(#port);
InetAddress group =
  InetAddress.getByName(
    "230.0.0.1");

sc.joinGroup(group);

sc.receive(in);
System.out.println(new String(msg));
```



Émetteur

```
sc = new DatagramSocket();
// ou new MulticastSocket
// si il est aussi récepteur
InetAddress group =
  InetAddress.getByName(
    "230.0.0.1");

byte msg[] =
  "Hello, World!!!".getBytes();

sc.send(
  new DatagramPacket(msg, 0
    msg.length, group, #port);
```

4.d. Sockets en mode multicast

Limiter la portée des messages multicast

En fixant le TTL (setTimeToLive(int))

- ✓ 0 : ne dépasse pas la machine
- ✓ 1 : ne dépasse pas le réseau local
- ✓ 127 : monde entier

Autres protocoles de diffusion basés sur le Multicast

- Jgroups, LRMP, JavaGroups…
- Fournissent d'autres propriétés comme
  - ✓ Fragmentation/défragmentation des messages (> 64ko)
  - ✓ Ordre garantie des messages
  - ✓ Notification d'arrivée et de départ de membres
  - ✓ …