

# Applications Serveur Java (Programmation concurrente distribuée)

IUT de Paris – S3

Année 2016-17

Jean-François Brette

# Cours 1 Parallélisme et concurrency

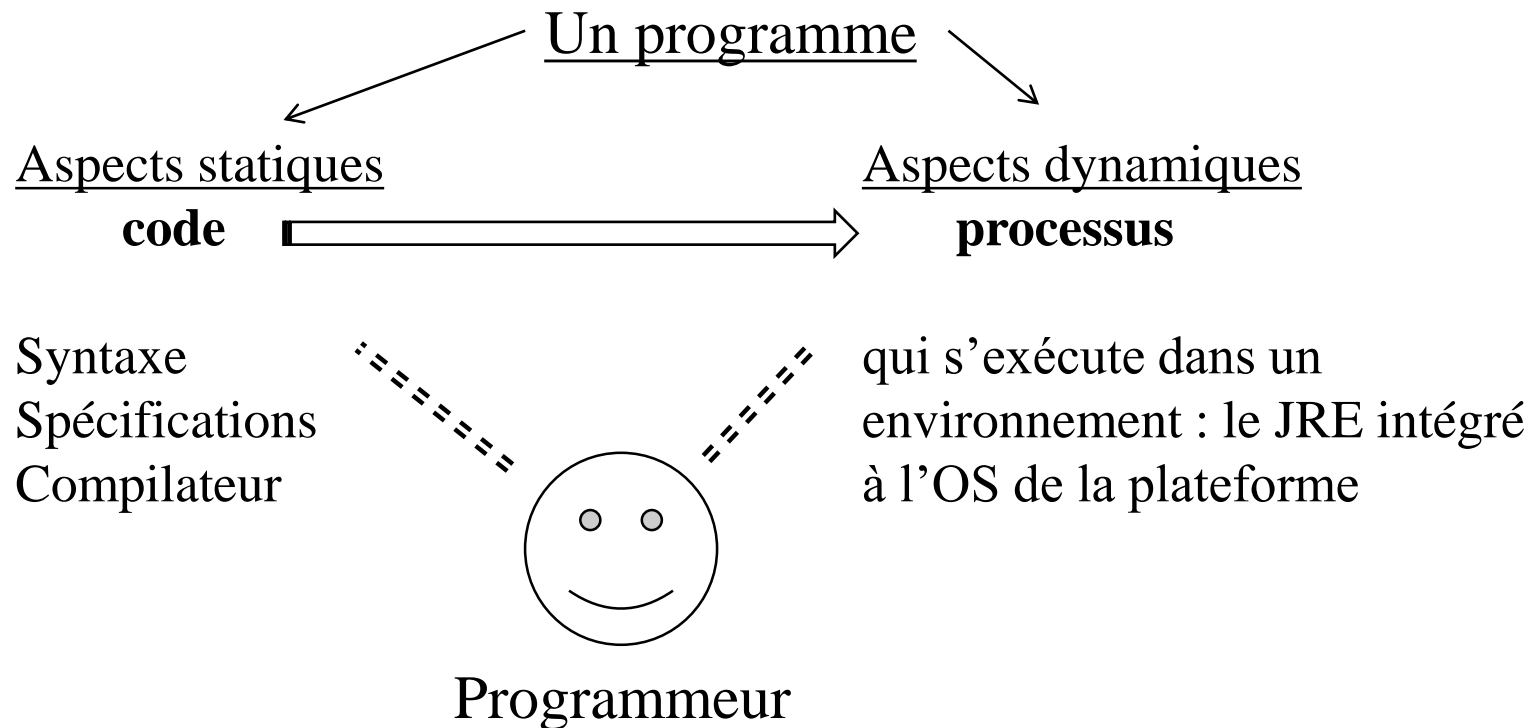
Processus légers (threads)

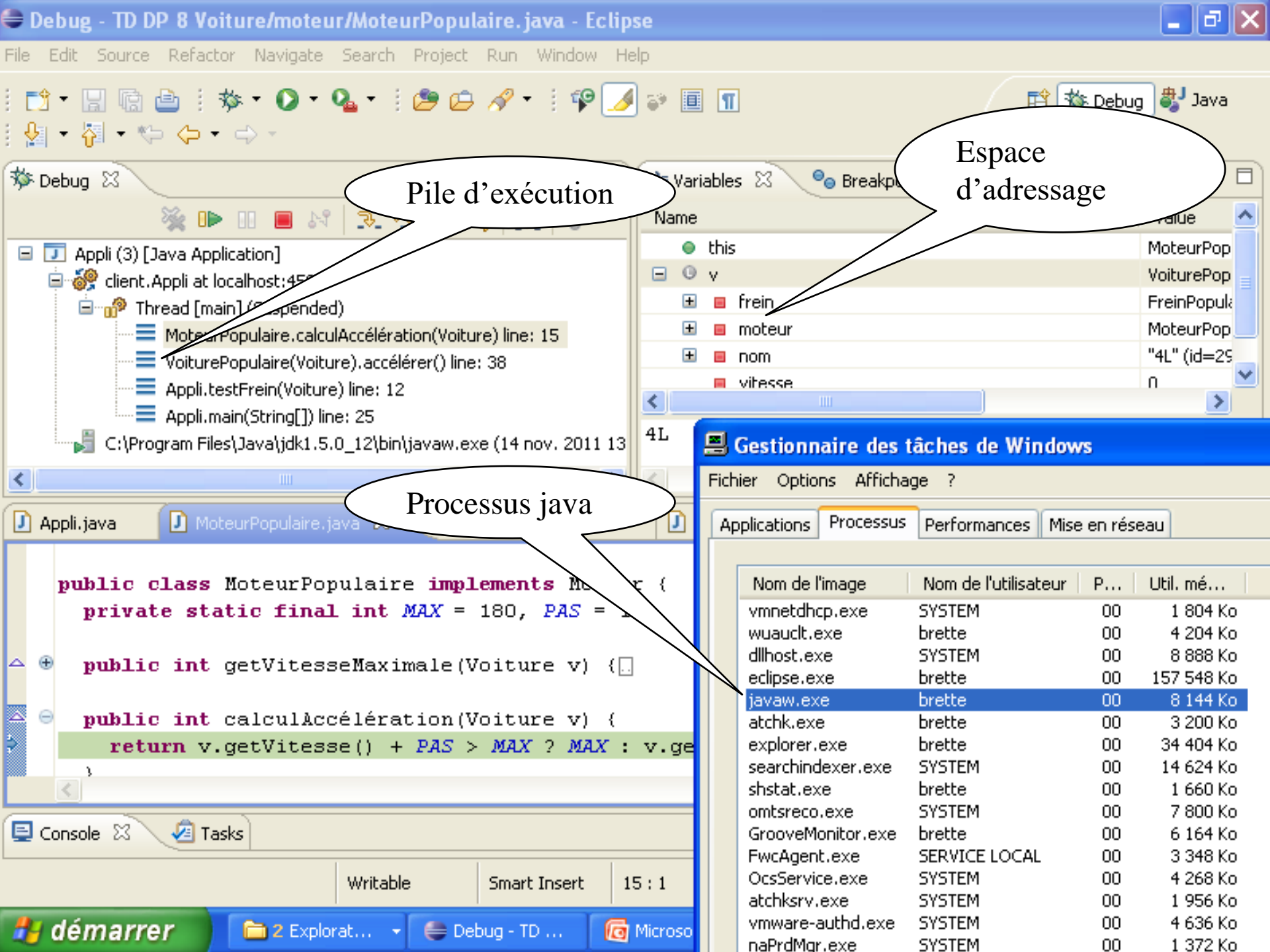
Programmation parallèle

Programmation concurrente

# Programme

## (aspects statiques et dynamiques)





# Processus vs thread

**Processus** (“lourd”) pile d’exécution propre

espace d’adressage propre

**Processus léger (thread)** créé au sein d’un processus lourd

pile d’exécution + esp. d’adressage propre

+ espace d’adressage partagé avec d’autres processus légers

Exécution en **parallèle** de plusieurs tâches (non bloquantes)

machine monoprocesseur => le scheduler

distribue le temps processeur (règle du tourniquet)

(ex.: anim.graph. + saisie texte + téléchargement d’une image)

Partage **concurrentiel** de données (ressources) entre plusieurs tâches

conflits, sections critiques, etc

# Exécuter une activité dans un thread

## Interface Runnable, classe Thread

```
public interface Runnable {  
    // runnable signifie exécutable dans un thread propre  
    public abstract void run ( );  
}
```

```
public class Thread implements Runnable {  
    // un thread = un processus léger  
    .....  
    public void run ( ) {.....};  
}
```

**Lancer un thread : le protocole start()**

# Lancer un exécutable (1)

## sous-classer Thread

```
public class Activité extends Thread {  
    /* Activité est un Runnable et a sa propre pile d'exécution par  
    héritage de Thread */  
    public Activité ( ) { this.start ( );}  
    .....  
    public void run ( ) { // description de l'activité..... };  
}
```

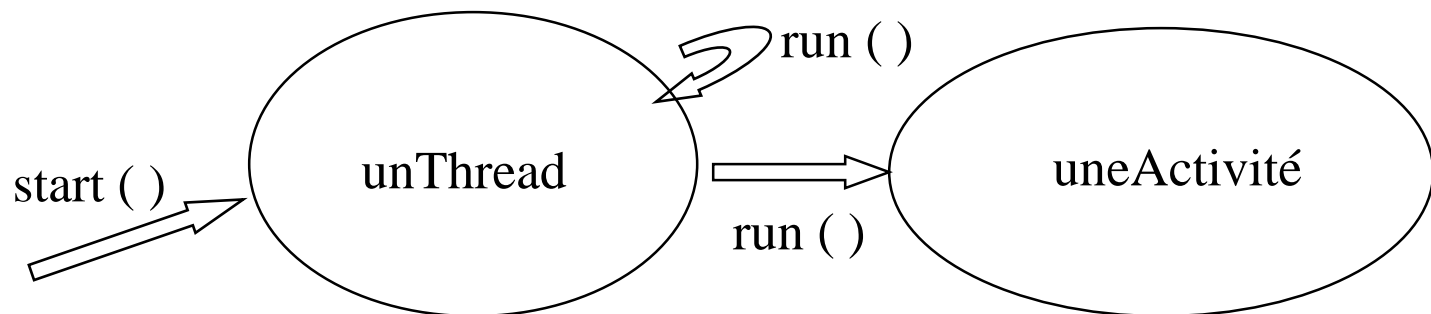
Code appelant : **new Activité();**



# Lancer un exécutable (2)

## implémenter Runnable

```
public class Activité implements Runnable {  
    /* Activité est un Runnable (explicitement qui doit être lié à un  
    Thread pour avoir sa propre pile d'exécution */  
    public Activité ( ) { (new Thread ( this ) ).start ( ); }  
    .....  
    public void run ( ) { // description de l'activité..... }  
}
```





# Classe java.util.Thread

```
public class Thread implements Runnable {  
    private Runnable target;  
        public Thread( Runnable r) {this.target = r;}  
        public void start() {.....}  
        public void run ( ) { if (target != null) target.run() ;};  
}
```

*uneVoiturePopu.calculAccélération()  
uneVoiturePopu.accelerer()  
uneActivite.run()  
unThread.run()*

pile d'exécution du thread

*Le fond de pile ne contient  
pas un main()*

# Quelle solution entre 1 et 2 ?

Question : veut-on définir une activité parallèle à d'autres  
(y compris éventuellement avec partage de données)

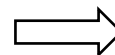
ou veut-on gérer un sous-type de processus léger (l'arrêter explicitement,  
le relancer, modifier sa priorité,...) ?



## **Programmeur d'application vs programmeur système**



Définit des activités parallèles  
Gère des données partagées



**implémenter Runnable**

# Attention à l'héritage !

```
public class Activité implements Runnable {  
    public Activité() {  
        new Thread(this).start();  
    }  
    public void run() { ... }  
}
```

Si nous dérivons cette classe :

```
public class ActivitéClient extends Activité {  
    private Client client;  
    public ActivitéClient (Client c) {  
        super();  
        this.client = c;  
    }  
    public void run() { this.client.getNom(); }  
}
```

le thread a démarré  
alors que ***this.client***  
n'est pas initialisée

risque de  
NullPointerException

# Lancer un thread

**Ne jamais permettre de lancer un thread non initialisé**

## **Solution 1 : interdire l'héritage**

```
public final class Activité implements Runnable {  
    .....  
}
```

La classe Actif  
n'est pas sous-  
classable

## **Solution 2 : séparer constructeur et lancement**

```
public class Activité implements Runnable {  
    public Activité (...) {...}  
}  
public void lancer() {new Thread(this).start();}  
}  
  
new Activité (...).lancer();
```

Le lancement doit  
être fait  
explicitement dans  
le code appelant

# Notion de « thread courant »

**Thread courant = le thread en train de s'exécuter**

**Accès au thread courant : méthode statique *currentThread()* de Thread**

```
public static Thread currentThread() {...}
```

**Méthodes de la classe Thread concernant le thread courant :**

public static void **dumpStack()** → recopie la pile d'exécution du thread courant

public static boolean **interrupted()** → test si le thread courant a été 'interrompu'

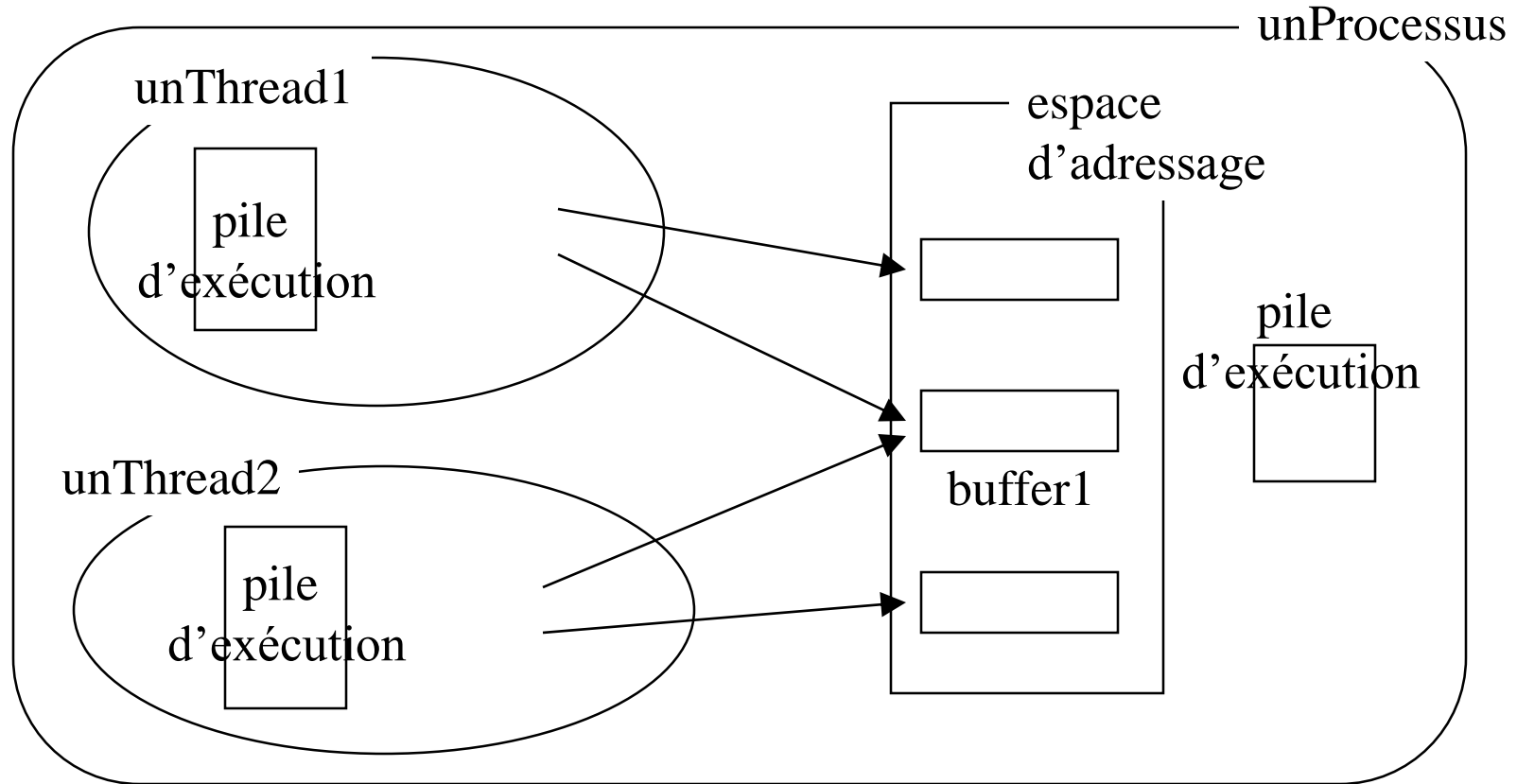
public static void **sleep**(long ms) throws InterruptedException

→ pause de *ms* millisecondes dans l'exécution du thread courant

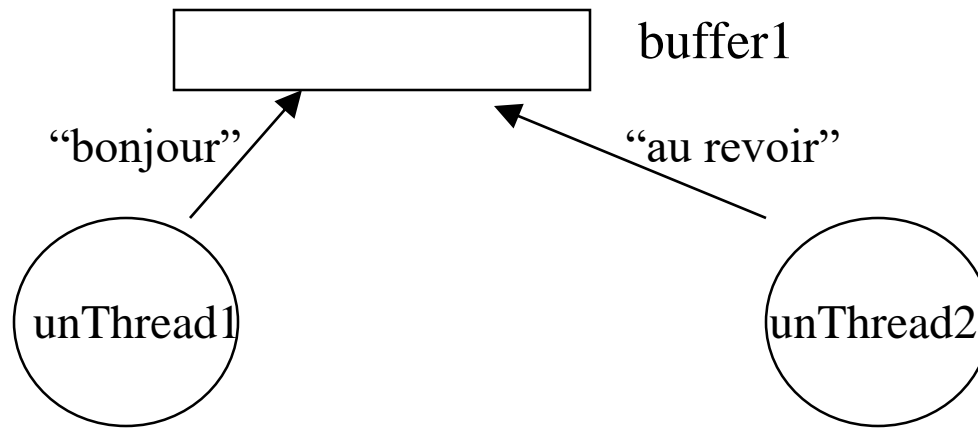
public static void **yield()**

→ suspend l'exécution du thread courant

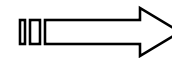
# Accès concurrentiel



# Accès concurrentiel (2)

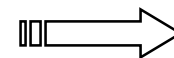


bonjour au revoir



section critiques + verrou  
(threads asynchrones)

bonjour au revoir  
au revoir bonjour



logique d'ordonnancement  
des threads  
(threads synchrones)

# Sureté : thread safety

Thread safety : un objet peut-il supporté d'être adressé en parallèle par plusieurs threads ?

Exemple : java.util.ArrayList n'est pas thread-safe

```
private int size;  
private E[] elementData;  
public void add(E element) {  
    elementdata [size] = element ;  
    size++;  
}
```

**section critique non verrouillée**

→ java.util.Vector

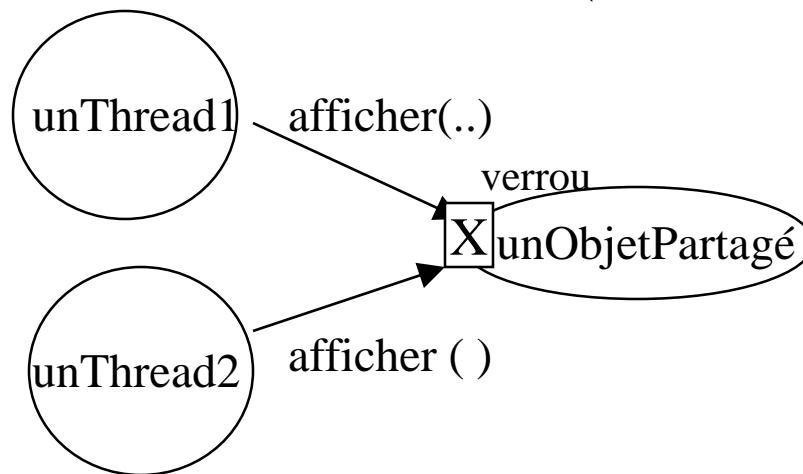


# Section critique + verrou

Eviter “bonaujo revuoir”  $\Rightarrow$  *synchronized*

Séquentialiser l'accès aux ressources partagées lorsqu'il y a un risque.  
Tout objet Java possède un verrou (moniteur de Hoare) et son accès peut donc être verrouillé.

*synchronized*  $\Rightarrow$  bloc de code où le parallélisme est interdit  
(= section critique)



```
class ObjetPartagé {  
    synchronized void afficher (..) {....}  
    synchronized void message2 {....}  
    .....  
}
```

# Exemple : une Fifo partagée

*(cf le code complet sur le serveur commun)*

Une file (Fifo)

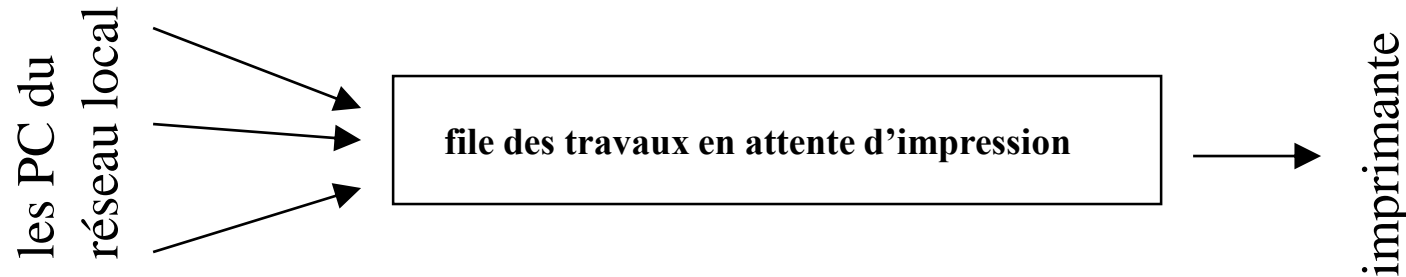
Des producteurs qui écrivent (*push*)

Des consommateurs qui lisent (*pop*)

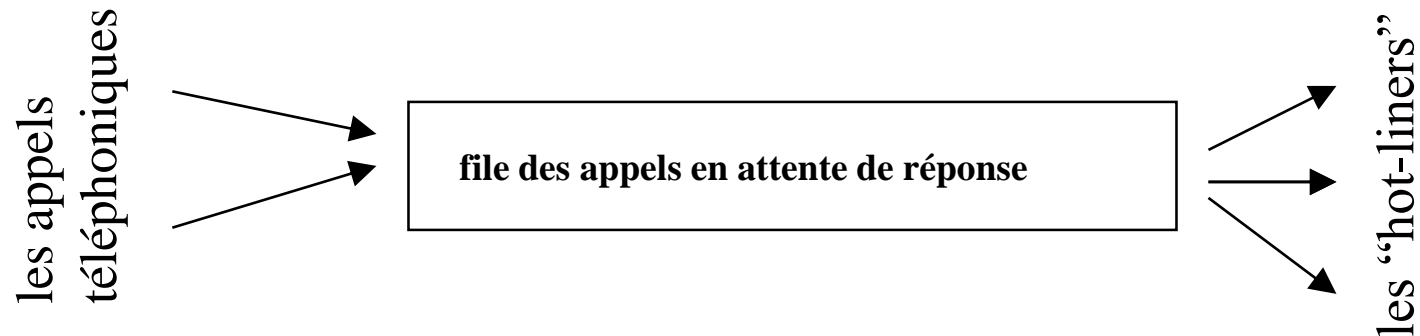


# Exemples de fifos partagées

Une fifo n-producteur 1-consommateur (spool d'imprimante)



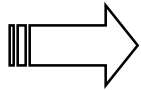
Une fifo n-producteur n-consommateurs (hot line)



## Séquentialiser les méthodes d'accès *pop* et *push*

```
public class Fifo <E> {.....  
    public synchronized void push (E anObject) {...}  
    public synchronized E pop ( ) {...}  
    .....  
}
```

- sécurité garantie (thread safety)
- pas très souple ni très efficace (toutes les sections critiques sont verrouillées quand on passe dans une)



- attention aux performances :

- \* faut-il verrouiller isEmpty() ?

- \* réduire la taille des blocs synchronisés

```
public void pop (E anObject) {... // section non-section critique  
    synchronized (this) { // section critique.....  
    }  
    ..... // section non-section critique  
} // fin méthode pop
```

# Verrou et thread courant

```
public class Activité implements Runnable {  
    private RessourcePartagée maRessource;
```

```
    public void run() {  
        ..... maRessource.action();  
    }  
}
```

```
public class RessourcePartagée {  
    public void action() {  
        synchronized(this) {  
            .....  
        }  
    }  
}
```

*maRessource.action()  
uneActivite.run()  
unThread.run()*

*Ici, le thread courant  
prend le verrou de la  
ressource partagée*

# Verrou et thread courant : la fifo

```
public class Producteur implements Runnable {  
    private Fifo<Client> maFile;  
    public void run() {  
        this.maFile.push(...);  
    }  
}
```

```
public class File<E> {  
    public void push(E objet) {  
        synchronized(this) {  
            .....}  
        }  
    }  
}
```

```
..... main.....  
File<Client> uneFile = new File<Client>();  
new Producteur(uneFile).lancer(); //pile d'exec1  
new Producteur(uneFile).lancer(); //pile d'exec2
```

*maFile.push(...)  
unProducteur1.run()  
unThread1.run()*

*pile d'exec 1*

*maFile.push(...)  
unProducteur2.run()  
unThread2.run()*

*pile d'exec 2*

*Deux threads **concurrents**  
pour l'exécution de push pour  
uneFifo.*

*Le premier thread qui rentre  
dans le bloc synchronized  
**verrouille** uneFifo*

# Verrou et moniteur

```
public class Producteur implements Runnable {  
    private Fifo<Client> maFile;  
    public void run() {  
        this.maFile.push(...);  
    }  
}
```

```
public class File<E> {  
    public void push(E objet) {  
        synchronized(this) {  
            .....  
        }  
    }  
}
```

```
..... main.....  
File<Client> uneFile = new File<Client>();  
new Producteur(uneFile).lancer();  
new Producteur(uneFile).lancer();
```

*maFile.push(...)*  
*unProducteur1.run()*  
*unThread1.run()*

*pile d'exec 1*

*maFile.push(...)*  
*unProducteur2.run()*  
*unThread2.run()*

*pile d'exec 2*

*unThread1 vient de rentrer dans le bloc : le moniteur de Hoare de maFile garde la référence de unThread1 qui désormais possède le verrou jusqu'à la sortie du bloc sync : aucun autre thread ne peut exécuter ce bloc*

## Performance : thread actif/en veille

- Contexte : un thread t1 attend un état particulier de la ressource
- Attente active sur la fifo : t1, en testant l'état de la ressource de façon répétée, empêche celui-ci d'être modifié
- t1 consomme du temps processeur sans succès et ralenti les threads qui pourraient le satisfaire!



## actif → veille

Lorsque le contexte (l'état de la ressource partagée) est défavorable, le thread qui possède le verrou le libère et est mis en veille (`wait( )` envoyé à la ressource verrouillée). Le moniteur de la ressource garde un lien vers ce thread mis en veille.

Le scheduler ne lui donne plus de temps processeur, sa pile d'exécution est arrêtée.

```
class RessourcePartagée {  
    synchronized ... methode1( ) {.....  
        while (conditionNonRemplie( )) {  
            // on endort le thread qui possède le verrou  
            this.wait( );  
        }  
        // suite du travail au réveil  
        .....  
    }  
}
```

## veille → actif

Plus tard, lorsque l'état de la ressource aura été modifié, on réveillera (notifyAll( ) envoyé à la ressource) tous les threads connus du moniteur de la ressource et, lorsque le scheduler lui donnera la main, l'un d'entre eux pourra reprendre le verrou et poursuivre son travail.

```
class RessourcePartagée {  
    .....  
    synchronized ... methode2 ( ) {.....  
        // on a changé l'état de l'objet  
        // on réveille le(s) thread(s) qui attendent le verrou  
        this.notifyAll( );  
    }  
}
```

**Attention : wait() et notifyAll() sont envoyées à une ressource verrouillée sinon → IllegalMonitorStateException**

# Exemple : envoi d'une demande et attente (active) de la réponse

```
public class ReceptionDemande implements Runnable {  
    private static Fifo fifo; // avec un set static  
    private Reponse laReponse;  
    public void reponse(Reponse r) {  
        this.laReponse = r;  
    }  
    public void run() {  
        ... construction d'un objet Demande (saisie/clavier,...)  
        fifo..push(laDemande);  
        // attente active de la réponse  
        while (this.laReponse==null) ;  
        System.out.println(laReponse);  
    }  
}
```

# Exemple : envoi d'une demande et attente (en veille) de la réponse

```
public class ReceptionDemande implements Runnable {  
    private static Fifo fifo; // avec un set static  
    private Reponse laReponse;  
    public void reponse(Reponse r) {  
        this.laReponse = r;  
        synchronized(this) {this.notifyAll();}  
    }  
    public void run() {  
        ... construction d'un objet Demande (saisie/clavier,...)  
        fifo..push(laDemande);  
        // attente en veille de la réponse  
        synchronized(this) this.wait() ;  
        System.out.println(laReponse);  
    }  
}
```

# Objet verrou explicite

```
public class ReceptionDemande implements Runnable {  
    private static Fifo fifo; // avec un set static  
    private Reponse laReponse;  
    // c'est le moniteur de Hoare de attente qu'on utilisera  
    private Object attente;  
    public void reponse(Reponse r) {  
        this.laReponse = r;  
        synchronized(attente) { attente.notifyAll(); }  
    }  
    public void run() {  
        ... construction d'un objet Demande (saisie/clavier,...)  
        fifo..push(laDemande);  
        // attente en veille sur verrou explicite  
        synchronized(attente) attente.wait() ;  
        System.out.println(laReponse);  
    }  
}
```

# Lecteur/écrivain

## Partage de documents

Contexte : plusieurs écrivains, plusieurs lecteurs d'un même document

Politique d'accès : soit 1 seul écrivain, soit plusieurs lecteurs

Demande de lecture : mise en attente si écrivain en cours

Demande d'écriture : mise en attente si écrivain ou lecteurs en cours

Fin de lecture ou d'écriture : réveil des lecteurs et écrivains en attente

# Lecteur/écrivain (code)

```
private boolean ecrivainEnCours;
private int lecteursEnCours;

public void lecture(...) throws InterruptedException {
    synchronized(this) { while (ecrivainEnCours) { wait(); }
                        lecteursEnCours++
    }
    // lecture hors section critique
    // d'autres lecteurs peuvent rejoindre....
    .....
    synchronized(this) { lecteursEnCours--;
                        this.notifyAll();
    }
} // fin lecture
```

# Lecteur/écrivain (code)

```
private boolean ecrivainEnCours;
private int lecteursEnCours;

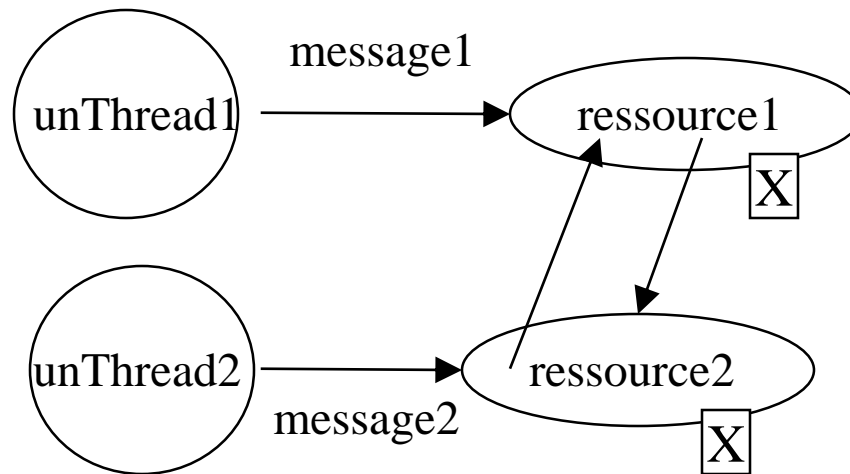
public void ecriture (...) throws InterruptedException {
    synchronized(this) { while (ecrivainEnCours || lecteursEnCours > 0)
                           {wait(); }
                          ecrivainEnCours = true ;
    }
    // écriture hors section critique
    .....
    synchronized(this) {ecrivainEnCours = false ;
                        this.notifyAll();
    }
} // fin écriture
```



# Vivacité : éviter les blocages

Verrou : risque de blocage (verrous mortels ou dead locks)

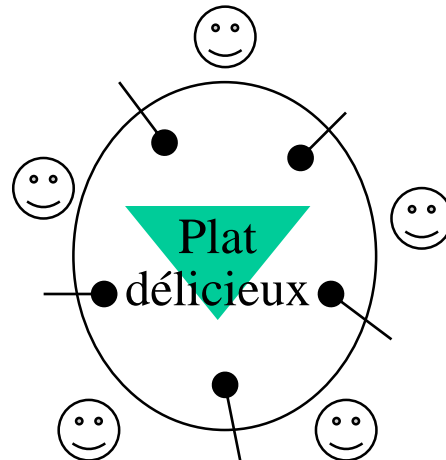
Cas le plus fréquent : interblocage à 2



Aucun moyen de sortir d'un blocage : risque majeur !

# Blocage : diner des philosophes

5 philosophes + table ronde + 5 cuillères + plat délicieux au milieu  
Pour manger : 2 cuillères (droite et gauche)



Chaque philosophe prend sa cuillère droite → interblocage

# Blocage simple : ordre du verrouillage

```
public class LeftRightDeadlock {  
    private final Object left = new Object();  
    private final Object right = new Object();  
    public void leftRight() {  
        synchronized(left) {  
            synchronized(right) {  
                action();  
            }  
        } // fin leftRight  
    public void rightLeft() {  
        synchronized(right) {  
            synchronized(left) {  
                autreAction();  
            }  
        } // fin rightLeft  
    }  
    ... suite de la classe
```

**Ne prenez jamais des verrous  
dans un ordre différent !**



# Blocage plus complexe : ordre du verrouillage

```
public void transferMoney(Compte débité, Compte crédité, int montant)
    throws CompteInsuffisantException {
    synchronized (débité) {
        synchronized (crédité) {
            if (débité.crédit() < montant)
                throw new CompteInsuffisantException (débité, montant);
            débité.retrait(montant);
            crédité.dépot(montant);
        }
    }
}
```

**transferMoney(compte1, compte2)**



unThread1

**transferMoney(compte2, compte1)**



unThread2

# Cours 2 Communication TCP par sockets

Communication entre machines

Sockets

# Communication entre machines

Bas niveau : sockets

Client/serveur : JDBC, servlets, JSP

Informatique distribuée:RMI, CORBA, EJB

# Rappels réseau

Couche la plus basse : IP Internet Protocol

Couche transport : TCP, UDP

Ports d'E/S

Sockets et client/serveur

# Couche transport : TCP, UDP

**Définit le mode de transmission des paquets de données**  
**2 options : Très fiable ou très rapide**

**TCP :** Transmission Control Protocol

Assure la réexpédition des données égarées

Le récepteur émet un accusé de réception pour chaque paquet

Transmission de fichiers dont la complétude doit être garantie

**UDP :** User Datagram Protocol

Pas d'AR, pas d'assurance de bonne transmission

Transmission de fichiers volumineux pour lesquels un pourcentage de perte d'informations est envisageable (images, vidéos,...)



# Sockets

**Socket : canal logique de communication entre 2 machines**

**Avantages :** les aspects *Système/Réseau* ne sont pas à gérer  
(décomposition du message en paquets, en tête de paquet,  
réexpédition en cas de pertes pour TCP, ...)

**Client/serveur :** une machine attend, l'autre a l'initiative

**Deux aspects :**

- le serveur attend les demandes sur un port défini
- un client s'est manifesté, une socket est créée en mode point à point entre le serveur et le client

# Client/serveur

Plusieurs clients peuvent être en liaison avec le serveur en même temps et sur le même port d'E/S sans risque de confusion

Multithreading coté serveur :

n clients  n objets communicants + 1 objet attendant

Opérations élémentaires liées aux sockets :

- s'attacher à un port d'E/S

- attendre les demandes de connexion émanant des clients

- accepter les connexions sur le port défini

- se connecter à un serveur

- envoyer, recevoir des données

- clôre une connexion

# java.net

Gestion des URL :

URL, URLConnection, URLEncoder, MalformedURLException

Datagrammes UDP :

DatagramPacket, DatagramSocket, MulticastSocket, ...

Socket à comportement non standard :

SocketImpl, SocketImplFactory, ...

Gestion des sockets :

InetAddress (adresse IP), ContentHandler (gestionnaire de contenu)

**Socket (pour les sockets) et ServerSocket (pour les serveurs)**

# La classe Socket

Une instance de la classe Socket permet de :

- se connecter à un serveur
- envoyer, recevoir des données
- clôre une connexion

## **Se connecter : instantiation**

### **Préciser l'adresse IP et le port d'E/S**

```
Socket maSocket = new Socket ("acsi.iut.univ-paris5.fr", 1234);  
Socket maSocket = new Socket ("192.93.28.9", 1234);
```

# La classe Socket (2)

## **Envoyer/recevoir des données**

### **Ecrire (resp. lire) dans le outputStream (resp. inputStream)**

```
BufferedReader socketIn =
```

```
    new BufferedReader (new InputStreamReader (maSocket.getInputStream ( ));
```

```
PrintWriter socketOut = new PrintWriter (maSocket.getOutputStream ( ), true);
```

```
String line = socketIn .readLine( );
```

```
socketOut.println (“OK”);
```

## **Clore une connexion**

```
maSocket.close ( );
```

# La classe ServerSocket

Une instance de la classe ServerSocket permet de :

- s'attacher à un port d'E/S

- attendre les demandes de connexion émanant des clients

- accepter les connexions sur le port défini

**S'attacher à un port d'E/S**

**Lors de l'instantiation**

```
ServerSocket monServeur = new ServerSocket (1234);
```

# Ecriture d'un serveur monoclient

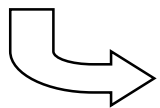
**Attendre des demandes de connexion et les accepter**

**La méthode `accept()`**

```
Socket socketCotéServeur = monServeur.accept ( );
```

Exemple de code correct pour le run du serveur ?

```
ServerSocket monServeur = new ServerSocket (1234);  
while (true) {  
    Socket socketCotéServeur = monServeur.accept ( );  
    ..... dialogue avec le client .....  
    socketCotéServeur.close ( );  
    }  
}
```



***un seul client à la fois !!!!***

# Ecriture d'un serveur n clients

**Il faut un thread de service pour chaque client  
+ un thread à l'écoute de nouveaux clients**

Exemple de code correct pour le run ( ) du serveur :

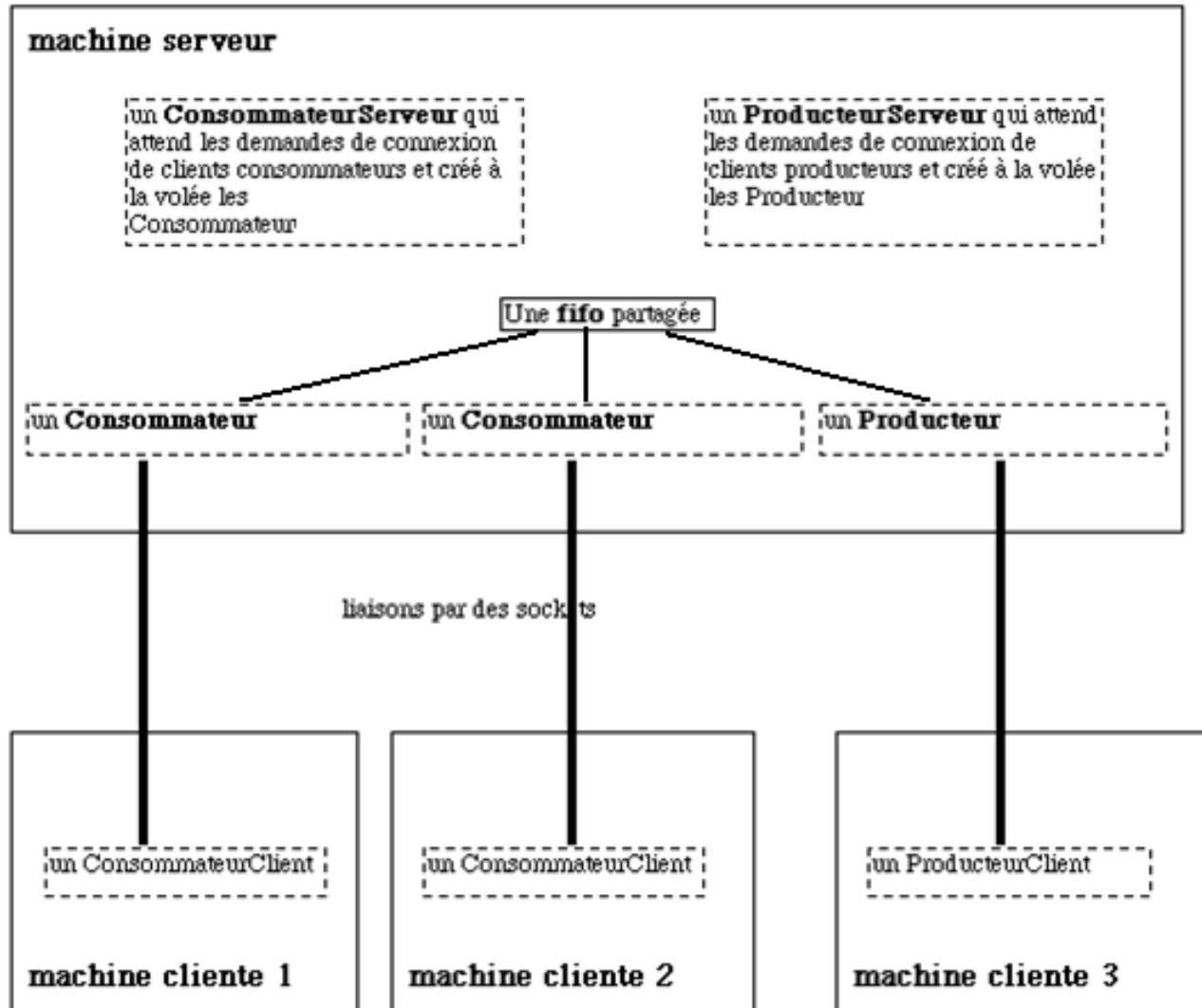
```
ServerSocket monServeur = new ServerSocket (1234);  
while (true) {  
    Socket socketCotéServeur = monServeur.accept ( );  
    new Service (socketCotéServeur, ...autres infos...);  
}
```

avec une classe Service implémentant Runnable :

```
class Service implements Runnable {.....  
    public void run ( ) {  
        ..... rendre le service au client .....  
    }  
}
```



# Application à une Fifo



# Consommateur

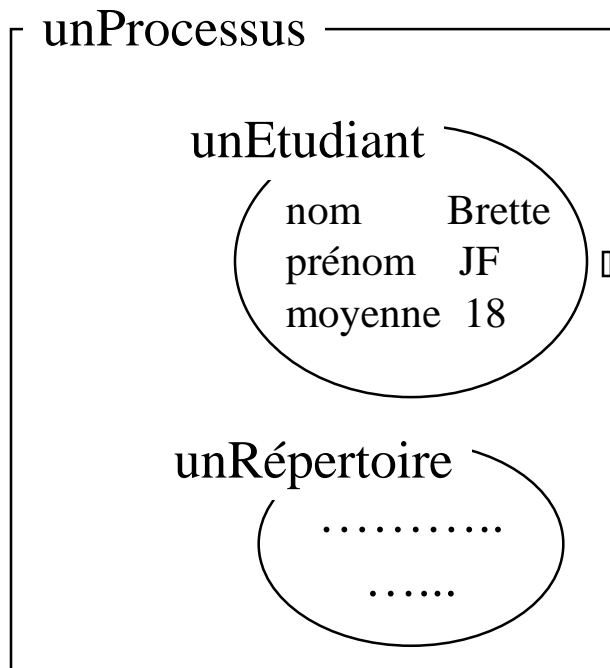
```
class Consommateur implements Runnable {  
    private Fifo laFifo;  
    private Socket socket;  
    Consommateur(Fifo f, Socket s) {  
        laFifo = f;  
        socket = s;  
        new Thread (this) . start( );  
    }  
    public void run ( ) { // il faudrait ajouter les try/catch  
        PrintWriter socketOut = .....  
        while (true)  
            socketOut.println (laFifo.pop ( ));  
    }  
    protected void finalize() throws Throwable {  
        socket.close( );}  
}
```

# Cours 3 Sérialisation

Objets persistants  
Sérialisation

# Objet persistant

**Préserver l'état d'un objet entre plusieurs exécutions de programmes**




**Extraire cet objet de son contexte  
(->fichier, transmission, ...)**

Copier les valeurs de tous ses attributs

Pb : jusqu'à quelle profondeur doit-on  
recopier ?

# Exemple : un répertoire d'adresses de serveurs

Une entrée du répertoire (i.e. un serveur)



nom	String
@IP	String
mesServices	liste (Vector) de Service

Avec Service



intitulé	String
noPort	int

```
class Répertoire { // un répertoire des serveurs
    private String propriétaire;
    private Vector <Serveur> mesServeurs;
    .....les méthodes
}
```

```
class Serveur { // un serveur
    private String nom;
    private String adIP;
    private Vector <Service> mes Services;
    ..... les méthodes
}
```

```
class Service { // un service d'un serveur
    private String intitulé;
    private int noPort;
    ..... les méthodes
}
```

**unRépertoire**

mesServeurs

nom	« Serveur1 »		
@IP	192.12.34.1		
mesServices	« Service1-1 »	1000	
	« Service1-2 »	1001	
	« Service1-3 »	1002	
nom	« Serveur2 »		
@IP	192.12.34.2		
mesServices	« Service2-1 »	2000	
	« Service2-2 »	2001	
	« Service2-3 »	2002	
etc			

Propriétaire Brette

**désérialisation**suite de bytes  
indépendants du contexte**sérialisation**  
(données+déclaration de classe)

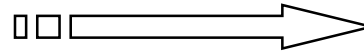
# Sérialisation

**Un objet qui est destiné à être persistant doit être sérialisable**

```
interface java.io.Serializable { }
```

```
class MonObjetPersistant implements java.io.Serializable {  
.....  
}
```

**java.io.ObjectOutputStream**  
**java.io.ObjectInputStream**



**sérialiser**  
**désérialiser**

➡ **On ne passe pas dans le constructeur lors de la désérialisation**



# Exemples de classes sérialisables

```
public  
class Vector <E> implements java.io.Serializable {  
.....  
}
```

```
public  
class HashMap<K,V> extends Dictionary implements java.io.Serializable {
```

```
public final  
class String implements java.io.Serializable {
```

**Number** et ses sous-classes (**Byte**, **Double**, **Float**, **Integer**, ...)

.....

# Exemples de classes non sérialisables

Les classes dont les instances sont liées à l'état d'une ressource  
(fichier ouvert, connexion par socket ou autre, processus, ... )

**Process**

**Thread**

**Socket**

**InputStream et ses sous-classes**

**OutputStream et ses sous-classes**

# Un exemple simple

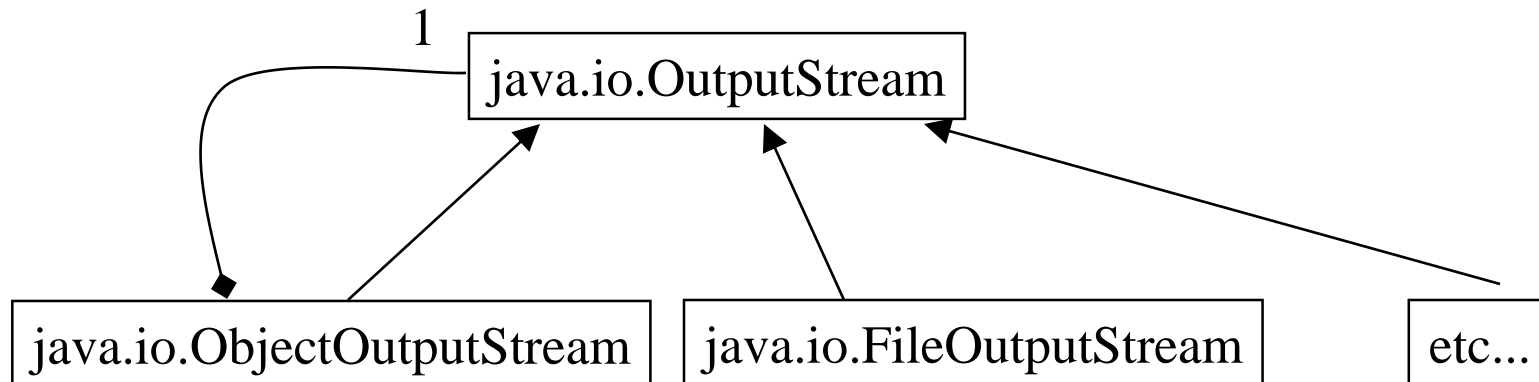
```
class Personne implements java.io.Serializable {  
    private String nom;  
    private String adresse;  
    ..... constructeurs .....  
    ..... accesseurs .....  
}  
..... utilisation pour écrire sur fichier.....  
Personne unePersonne = new Personne (“Brette”, “.....”);  
String fname = .....  
FileOutputStream fout = new FileOutputStream (fname);  
ObjectOutputStream out = new ObjectOutputStream (fout);  
out.writeObject(unePersonne); // c’est fait !
```

.....**utilisation pour envoyer dans une socket.....**

```
Personne unePersonne = new Personne (“Brette”, “.....”);
```

```
Socket s = new Socket (IPmachine, portES);
```

```
ObjectOutputStream out =  
                  new ObjectOutputStream (s.getOutputStream());  
out.writeObject(unePersonne); // c’est fait !
```



# Classe

## java.io.ObjectOutputStream

```
public class ObjectOutputStream extends OutputStream  
implements ObjectOutput  
{.....  
    private OutputStream out;  
    .....  
    public ObjectOutputStream(OutputStream out) throws IOException {  
        // unique constructeur  
        .....  
    }  
    public final void writeObject(Object obj)  
        throws IOException {  
        .....  
    }  
    .....  
}
```

# Interface java.io.ObjectOutput

```
public interface ObjectOutput extends DataOutput  
{  
  
public void writeObject(Object obj) throws IOException;  
  
public void write(byte[] b) throws IOException;  
.....  
public void flush() throws IOException;  
public void close() throws IOException  
}
```

# Classe java.io. ObjectInputStream

```
public class ObjectInputStream extends InputStream implements ObjectInput
{
    public final Object readObject()
        throws ClassNotFoundException, IOException {
        .....
    }
    .....
}
```

**InvalidClassException** – Pb avec une classe utilisée pour la serialisation.

**StreamCorruptedException** - Information de controle inconsistent.

**OptionalDataException** - Primitive data au lieu d'objets dans le buffer

**Sous-classes de ObjectStreamException**

# Exemple d'utilisation (test)

**// un ObjectInputStream lié au fichier**

```
ObjectInputStream in = new ObjectInputStream  
    (new FileInputStream (fileName));
```

**// un ObjectOutputStream lié à la socket cliente**

```
ObjectOutputStream out = new ObjectOutputStream  
    (socketClient.getOutputStream());
```

**// boucle de lecture du fichier**

```
while (.....) {  
    Repertoire unRep = (Repertoire) in.readObject( );  
    System.out.println (unRep.proprietaire());  
    out.writeObject(unRep );  
}
```



# Modificateur *transient*

Permet d'indiquer au processus de sérialisation que la donnée ne devra pas être prise en compte

```
class Truc implements Serializable {  
    private String nom;  
    private Vector <Serveur> mesServeurs;  
    private transient ServerSocket unServeur;  
    .....  
}
```

```
unObjectOutputStream.writeObject ( unTruc ) ;  
// sans transient, erreur à l'exécution
```

# Sérialisation contrôlée

Permet de programmer soi même un mécanisme de sérialisation ad hoc.

Il faut alors implémenter dans la classe des objets à sérialiser  
par ex. Repertoire) les deux protocoles

```
private void readObject(java.io.ObjectInputStream stream)  
    throws IOException, ClassNotFoundException
```

```
private void writeObject(java.io.ObjectOutputStream stream)  
    throws IOException
```

# Contrôle de cohérence des données récupérées

**Permet de vérifier que des contraintes sur les champs sont respectées**

Ex.: les serveurs récupérés doivent avoir une adresse IP

--> *méthode validateObject( )*

```
class repertoire ..... {
```

```
.....
```

```
public void validateObject ( ) throws InvalidObjectException {
```

```
    for (Serveur serveur : mesServeurs)
```

```
        if (serveur.getAdIP( ) == null) throw new InvalidObjectException
```

```
            (“un des serveurs n’a pas d’adresse IP”+serveur.toString());
```

```
}
```