Applications Serveur Java (Programmation concurrente distribuée)

IUT de Paris – S3
Année 2017-18
Jean-François Brette

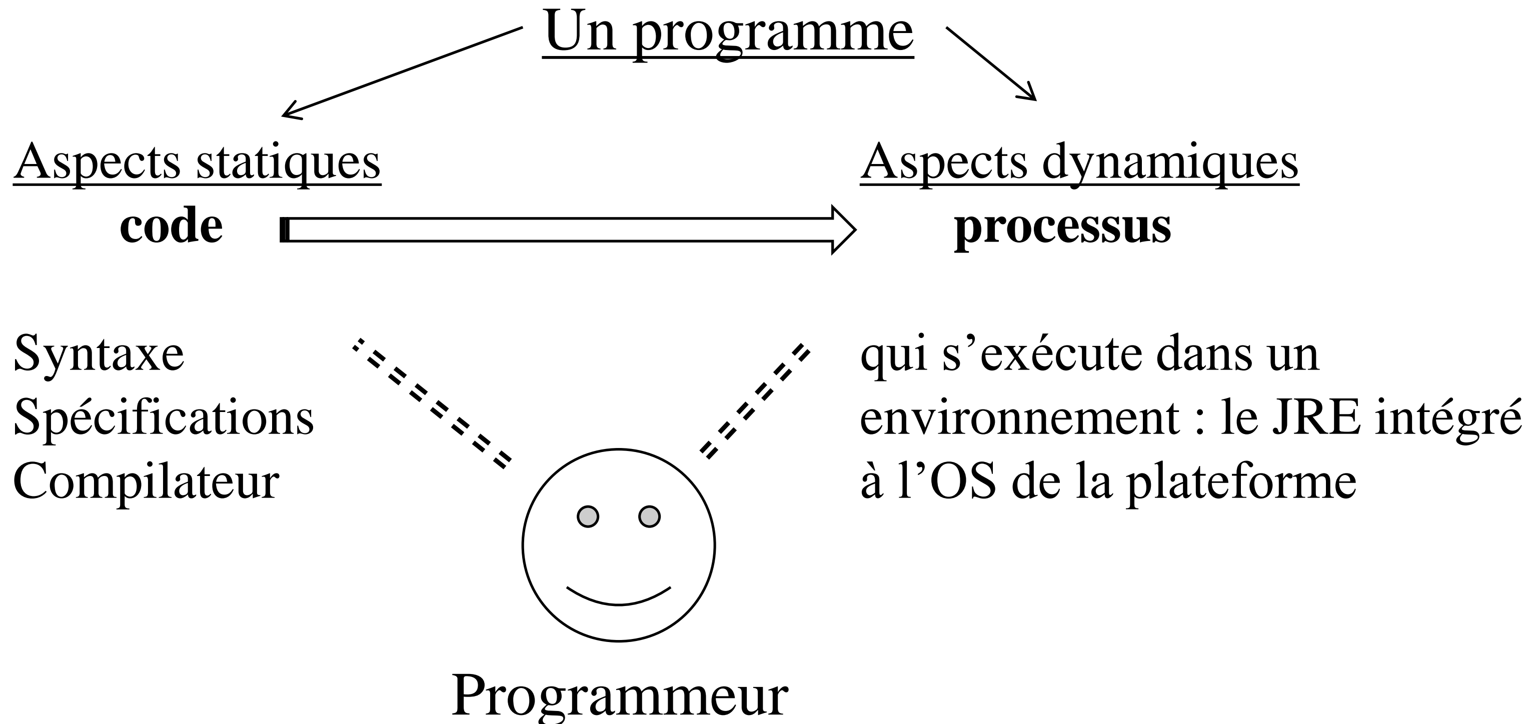
Cours 1 Parallélisme et concurrence

Processus légers (threads)

Programmation parallèle

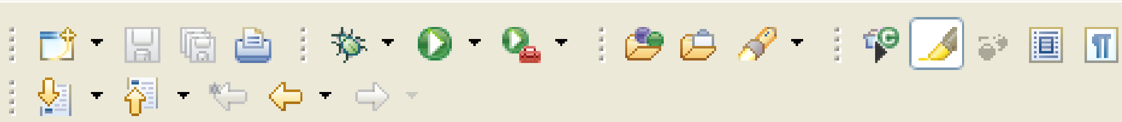
Programmation concurrente

Programme (aspects statiques et dynamiques)



Debug - TD DP 8 Voiture/moteur/MoteurPopulaire.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help



Debug Java

Pile d'exécution

Espace d'adressage

Debug

Appli (3) [Java Application]
client.Appli at localhost:45
Thread [main] (Suspended)
MoteurPopulaire.calculAccélération(Voiture) line: 15
VoiturePopulaire(Voiture).accélérer() line: 38
Appli.testFrein(Voiture) line: 12
Appli.main(String[]) line: 25
C:\Program Files\Java\jdk1.5.0_12\bin\javaw.exe (14 nov. 2011 13

Variables Breakp

Name	Value
this	MoteurPop
v	VoiturePop
frein	FreinPopula
moteur	MoteurPop
nom	"4L" (id=29
vitesse	0

4L

Processus java

Appli.java MoteurPopulaire.java

```
public class MoteurPopulaire implements Mo   r {
  private static final int MAX = 180, PAS = 1

  public int getVitesseMaximale(Voiture v) {

  public int calculAccélération(Voiture v) {
    return v.getVitesse() + PAS > MAX ? MAX : v.ge
  }
```

Console Tasks

Writable Smart Insert 15 : 1

démarrer 2 Explorat... Debug - TD ... Microso

Gestionnaire des tâches de Windows

Fichier Options Affichage ?

Applications Processus Performances Mise en réseau

Nom de l'image	Nom de l'utilisateur	P...	Util. mé...
vmnetdhcp.exe	SYSTEM	00	1 804 Ko
wuauclt.exe	brette	00	4 204 Ko
dllhost.exe	SYSTEM	00	8 888 Ko
eclipse.exe	brette	00	157 548 Ko
javaw.exe	brette	00	8 144 Ko
atchk.exe	brette	00	3 200 Ko
explorer.exe	brette	00	34 404 Ko
searchindexer.exe	SYSTEM	00	14 624 Ko
shstat.exe	brette	00	1 660 Ko
omtsreco.exe	SYSTEM	00	7 800 Ko
GrooveMonitor.exe	brette	00	6 164 Ko
FwcAgent.exe	SERVICE LOCAL	00	3 348 Ko
OcsService.exe	SYSTEM	00	4 268 Ko
atchksrv.exe	SYSTEM	00	1 956 Ko
vmware-authd.exe	SYSTEM	00	4 636 Ko
naPrdMgr.exe	SYSTEM	00	1 372 Ko

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()

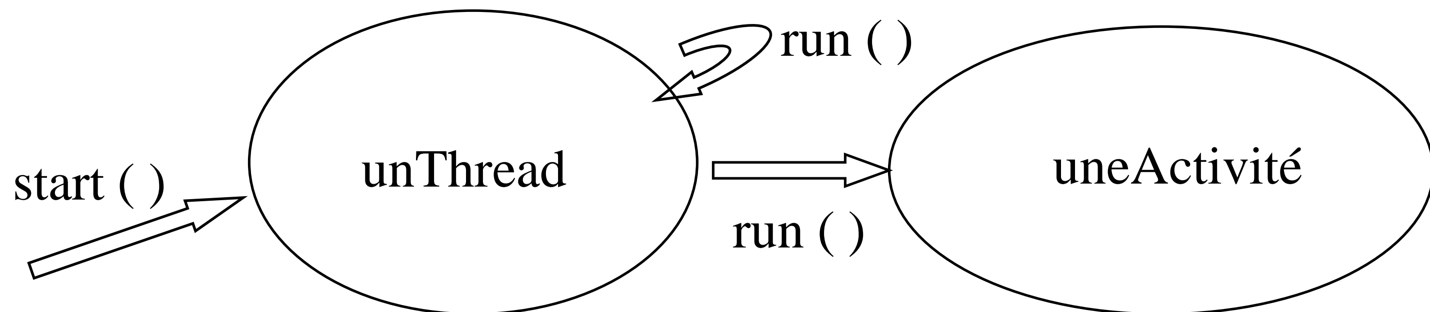
Lancer un exécutable (1) sous-classer Thread

```
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()

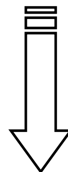
pile d'exécution du thread

Le fond de pile ne contient pas un main()

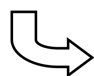
Quelle solution entre 1 et 2 ?

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

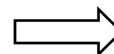
ou veut-on gérer un sous-type de processus léger (l'arrêter explicitement, le relancer, modifier sa priorité,…) ?



<u>Programmeur d'application</u> vs programmeur système



Définit des activités parallèles
Gère des données partagées



<u>implémenter Runnable</u>

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(); }
}
```

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

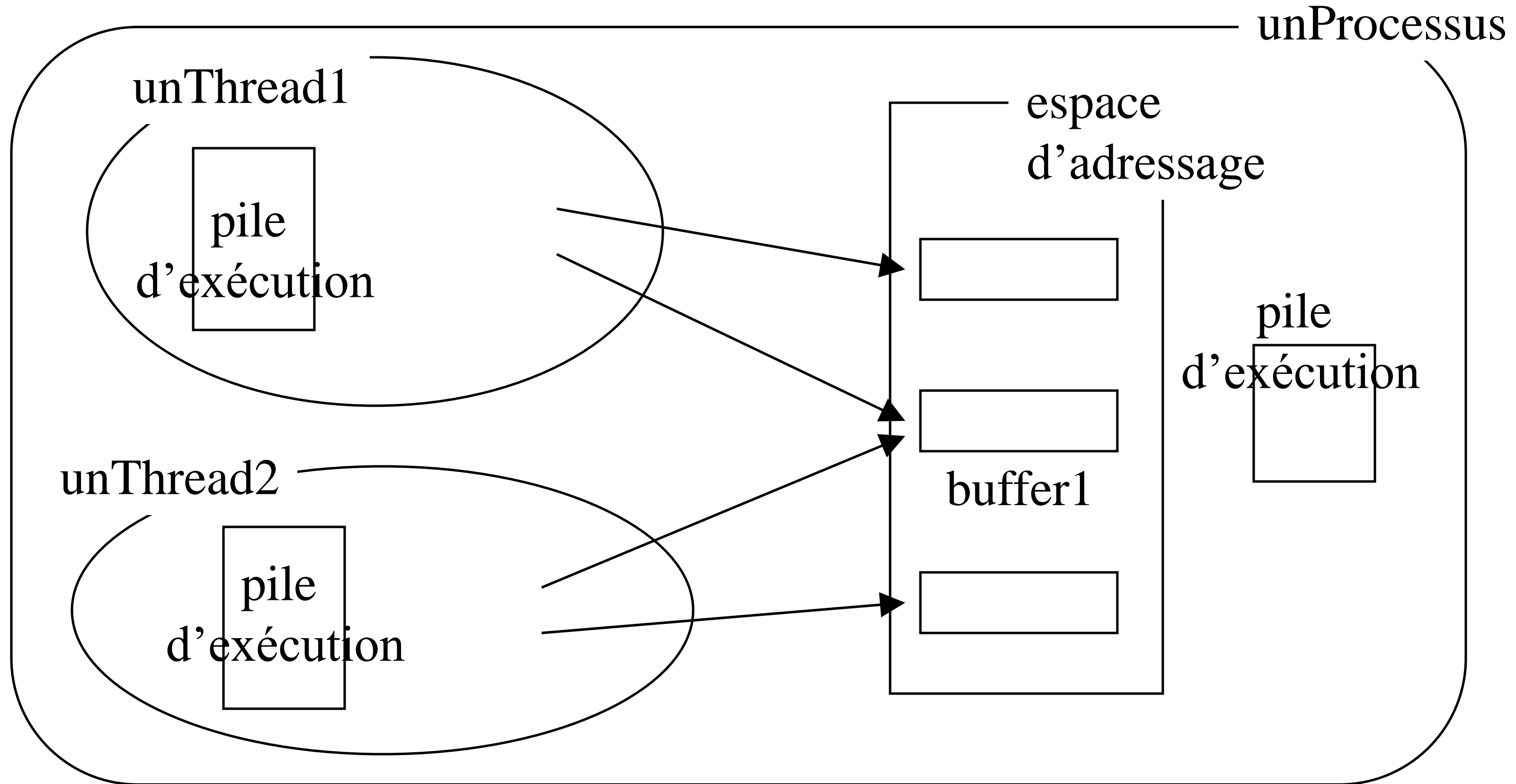
Accès au thread courant : méthode statique *currentThread()* de Thread

public static Thread currentThread() {…}

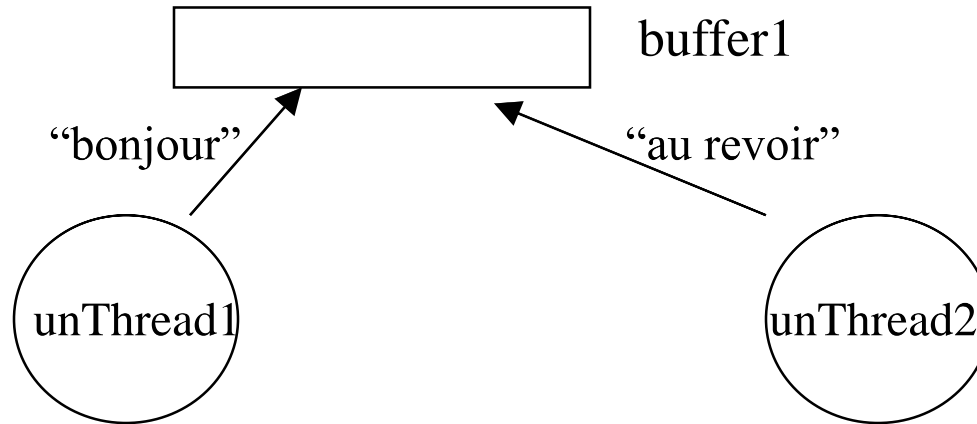
Méthodes de la classe Thread concernant le thread courant :

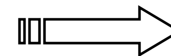
public static void **dumpStack**() → recopie la pile d'exécution du thread courant
public static boolean **interrupted**() → test si le thread courant a été 'interrompu'
public static void **sleep**(long ms) throws InterruptedException
→ pause de *ms* millisecondes dans l'exécution du thread courant
public static void **yield**()
→ suspend l'exécution du thread courant

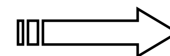
Accès concurrentiel



Accès concurrentiel (2)



bonaujo revuroir  section critiques + verrou (threads asynchrones)

bonjourau revoir
au revoirbonjour  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++;
}
```

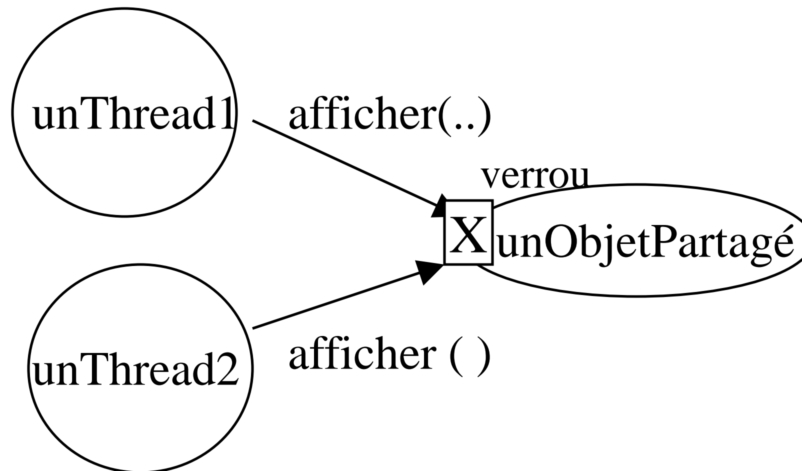
section critique non verrouillée (elementdata [size] = element ; size++;)

→ java.util.Vector

Section critique + verrou

Eviter "bonaujo revuroir" ⇨ *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 ⇨ 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*)

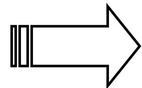
Exemples de fifos partagées

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 synchronised

```
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
```

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 ecrivain 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
```

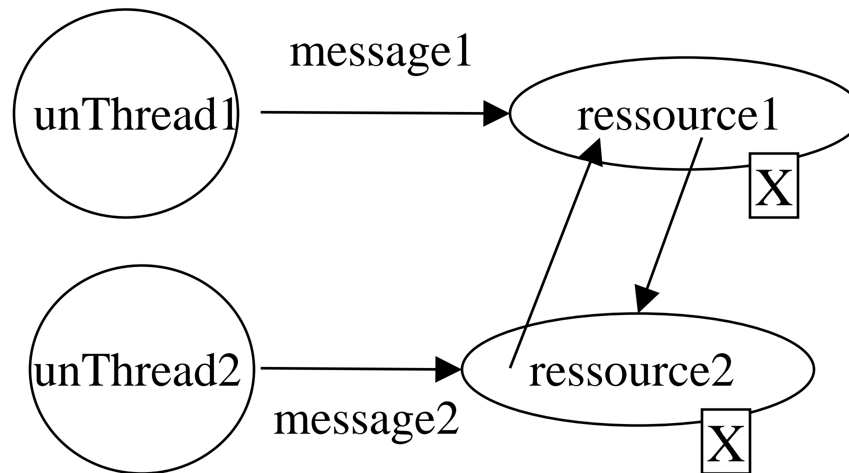
Lecteur/écrivain (code)

```
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

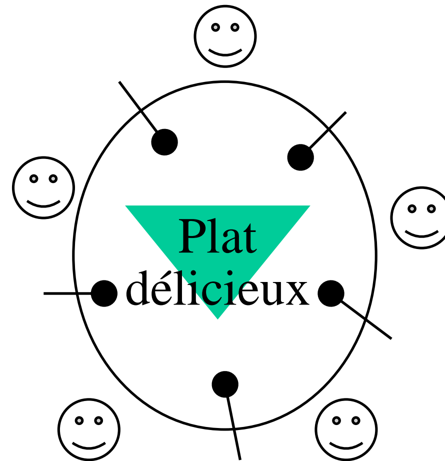
Verrou : risque de blocage (verrous mortels ou dead locks)
Cas le plus fréquent : interblocage à 2



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) **transferMoney(compte2, compte1)**

unThread1 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 :

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
- clore 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 :

InetAdress (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
clore 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

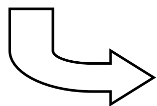
Attendre des demandes de connexion et les accepter

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

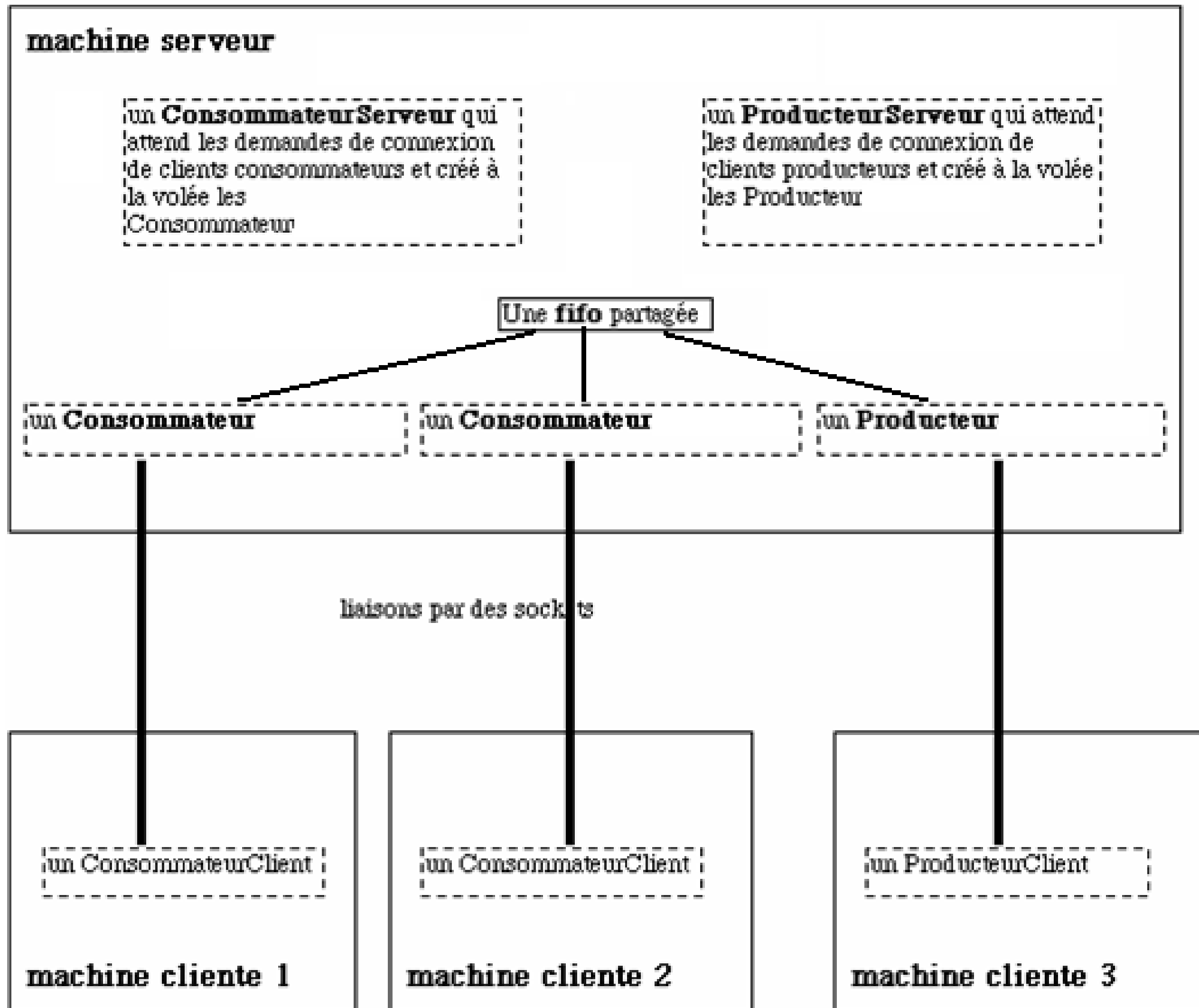
Exemple de code correct pour le run ( ) du serveur :

```
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( );}
}
```