

# Applications réseau

## Cours 5

Slides : Damien Imbs

Corentin Travers

`corentin.travers@univ-amu.fr`

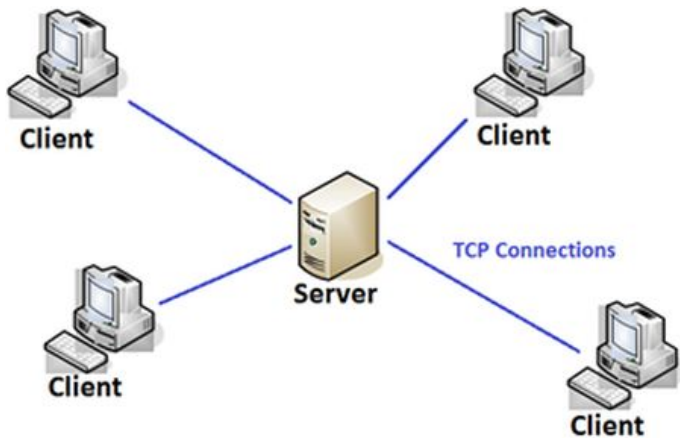
AMU - L3 info

2021-2022

# Clients multiples

# Communication Clients Serveur

Connexions multiples avec un seul serveur



# Communications Clients Serveur

Comment implémenter le serveur ?

- Mono-thread : un seul fil d'exécution
  - Selector pour des I/O multiplexées
- Multi-thread : un thread par connection
  - Si nombre de connexions supérieur au nombre de threads, réutiliser les threads terminés
- Hybride :
  - Plusieurs threads (nombre borné)
  - Chaque gère prend plusieurs clients (multiplexage avec Selector)

# Communication Clients Serveur

## Optimisation

Le logiciel côté serveur doit

- Minimiser le temps de réponse aux clients
- Minimiser le nombre de connexions perdu
- Maximiser le throughput pour le serveur
- Utiliser le moins de ressources possible (mémoire, débit)
- Équilibrer la charge parmi les processus/threads

L'efficacité est importante ; chaque microseconde perdue s'accumule avec le temps

Comment réduire le temps pour servir chaque requête ?

# Communication Clients Serveur

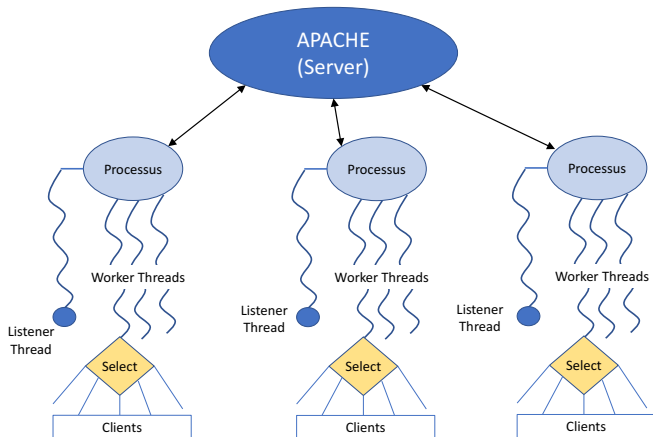
## Efficacité

Choisir la bonne architecture pour le serveur, selon

- les capacités du système (processeur, mémoire)
- la quantité de connexions attendue.

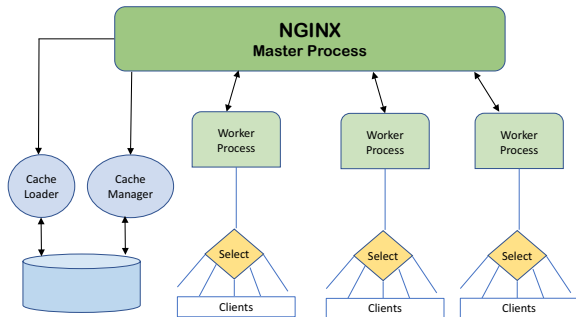
Quelles architectures sont utilisées en pratique ?

# Exemple : Serveur Apache



Un thread **Listener** par processus pour accepter les connexions clients et ensuite les distribuer parmi les threads **worker**

## Exemple : Serveur Nginx



Très peu de processus, mais chacun prend un grand nombre de clients (multiplexés). Répartition de charge efficace.

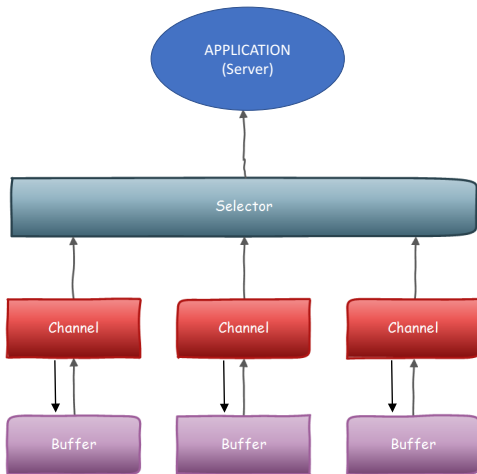


# Communication Efficace

**Java NIO** a été introduit avec Java 1.4 pour avoir des I/O plus efficaces.

- I/O asynchrone : pendant le transfert, le thread n'est pas bloqué
- Utilise les commandes natives de l'OS pour un transfert rapide des données.
- Transfère des blocs de données (au lieu de caractère par caractère avec les Java Streams)
- Les buffers NIO utilisent de l'espace mémoire en dehors de la JVM qui est accessible directement par l'OS.

# Rappel : Java NIO



# Communication Classiques (sans NIO)

```
Socket client1 = ServerSoc.accept();  
input1 = client1.getInputStream();  
// Recevoir des données de clients  
Byte b = input1.read();  
while (b >= 0) {  
    // Traitement de données...  
    b = input1.read();  
}
```

## Communication avec Java NIO

```
channel1 = ServerSocketChannel.accept();
channel1.configureBlocking(false);
ByteBuffer buffer =
    ByteBuffer.allocate(128);
channel1.read(buffer); // Non-bloquant
// ... Quand les données sont prêtes
buffer.flip();
while (buffer.hasRemaining()) {
    b = buffer.get();
    // Traitement des données...
}
```

# Java NIO : Channels et Buffers

Pourquoi utiliser les Buffer ?

(au lieu d'une communication directe avec la socket client)

- I/O Asynchrone entre Client Channel et le Buffer  
Attention : Il faut mettre le Channel en mode non-bloquant  
(`configureBlocking(false)`)
- I/O efficace vers le Buffer (directement par le système d'exploitation)  
Attention : il faut utiliser `allocate()` pour réserver la mémoire nécessaire.

# Java NIO Buffers

Classe principale : ByteBuffer

Les methodes principales

- Creation : allocate(taille)
- Ecriture : put() vs Channel.read()
- Lecture : get() vs Channel.write()
- Reinitialize : Clear()

## Exemple : NIO Buffer

```
buffer = ByteBuffer.allocate(128);\n// Lire donnée envoyé par client socket\n\nChannel1.read(buffer);\nbuffer.flip();\nwhile (buffer.hasRemaining())\n    System.out.print( buffer.get());\nbuffer.clear();\n\n//Envoyer données vers client socket\n\nwhile((b=System.in.get())!=null)\n    buffer.put(b);\nbuffer.flip();\nChannel2.write(buffer);
```

# Java NIO Buffers

Les paramètres : Position, Limit, Capacity, Mark

- Au début : Position=0, Limit=Capacity ;
- Après écriture : Position = 1+ dernier octet écrit.
- Utiliser flip() pour changer Position à zero et changer Limit à la dernière valeur de Position.
- Methodes rewind() et compact()
- Methodes mark() et reset()
- On peut accéder et changer directement les valeurs de Position, Limit, Capacity.



# Java NIO Channels

Les channels sont bidirectionnels (un seul channel par socket, au lieu d'utiliser inputStream, outputStream)

- Serveur TCP : Classe ServerSocketChannel
  - Création : `ServerSocketChannel.open()`
  - Associer à une adresse : `bind(InetSocketAddress)`
  - Écouter (attendre des requêtes) : `accept()`
- Client TCP : Classe SocketChannel
  - Méthodes :
    - `open()`, `connect()`, `read()`, `write()`

Un channel correspond à un socket (client / serveur)

## Exemple : TCP Socket Channels

// Côté Serveur :

```
ServerSocketChannel ssc  
    = ServerSocketChannel.open();  
ssc.configureBlocking(false);  
ssc.bind(new InetSocketAddress(host, port));  
SocketChannel c1 = ssc.accept();  
buffer1 = ByteBuffer.allocate(64);  
c1.read(buffer1);
```

## Exemple : TCP Socket Channels

// Côté Client :

```
SocketChannel soc = SocketChannel.open();  
saddr = new InetSocketAddress(host, port);  
soc.connect(saddr);  
buffer = ByteBuffer.allocate(16);  
buffer.put("Hello".getBytes());  
buffer.flip();  
soc.write(buffer);
```

# Java NIO Channels

- UDP socket : Classe DatagramChannel

Méthodes :

- send(buffer, socketAddress)
- receive(buffer)

- SelectableChannel :

- Type de channel qui peut être multiplexé pour les I/O
- Inclus : ServerSocketChannel, SocketChannel, DatagramChannel
- Méthode : configureBlocking(Boolean)  
( Mettre en mode bloquant ou non-bloquant )

# Plusieurs Buffers par Channel

Un channel peut écrire dans plusieurs Buffers

- Interface `ScatteringByteChannel`
- Méthode : `read(ByteBuffer[] data)` ;
- Etant donné un tableau de Buffers, écrit dans le 1er Buffer jusqu'à Limit, écrit ensuite dans le 2ème Buffer etc.

# Plusieurs Buffers par Channel

Un channel qui accumule les données de plusieurs buffers

- GatheringByteChannel
- Méthode : `write(ByteBuffer[] sources)`
- Le contenu du 1er Buffer est écrit dans le Channel, ensuite celui du 2ème Buffer etc.

Cette interface est implémentée par `FileChannel`, `DatagramChannel`, `SocketChannel`

# Java NIO Selector

I/O Multiplexé :

- Classe Selector
- Classe SelectionKey
- Enregistrer les Channels avec Selector  
Channel.register()
- Chaque association est identifiée avec un SelectionKey
- Ajouter les attachements avec SelectionKey  
SelectionKey.attach(), SelectionKey.attachment()

# Java NIO Selector

## Classe Java.nio.Selector

- Un objet Selector contient :
  - une liste de keys à attendre
  - une liste de keys sélectionnées.
- Chaque key correspond à un channel et à un événement associé.
- ServerSocketChannel  
Seule option : SelectionKey.OP\_ACCEPT
- SocketChannel, DatagramChannel  
Options : SelectionKey.OP\_READ,  
SelectionKey.OP\_WRITE, SelectionKey.OP\_CONNECT



# Enregistrement avec Selector

Creation :

```
Selector selector = Selector.open();
```

Enregistrement pour accept() par ServerSocketChannel ssc

```
ssc.register(selector, SelectionKey.OP_ACCEPT);
```

Enregistrement pour lecture (read) par SocketChannel client1

```
client1.register(selector, SelectionKey.OP_READ);
```

Enregistrement pour l'écriture (write) par SocketChannel c2

```
c2.register(selector, SelectionKey.OP_WRITE);
```

# Enregistrement et Attachement

## Classe SelectionKey

- On peut attacher un objet à une Key.

```
SelectionKey sk = ssc.register(  
    selector, SelectionKey.OP_ACCEPT);  
sk.attach(objet);
```

- Attachement : 3ème paramètre de la méthode register()

```
c1.register(selector, SelectionKey.OP_READ, objet);
```

- L'attachement peut être utilisé pour ajouter des infos (e.g. instant de l'enregistrement), ou pour ajouter un thread qui doit gérer l'événement.

```
SelectionKey key = ssc.register(  
    selector, SelectionKey.OP_ACCEPT, objet);  
Objet = key.attachment();
```

# I/O Multiplexées avec Selector

Méthode principale : `Selector.Select()`

```
int num = selector.select();  
/* attend les événements */  
Iterator<SelectionKey> keys =  
    selector.selectedKeys().iterator();  
while(keys.hasNext()) {  
    // Traiter l'événement ...  
}
```

Comment savoir quel type d'événement ?

# SelectionKey

SelectionKey sk = keys.next() ;

- sk.isAcceptable() == true  
Si l'événement est OP\_ACCEPT
- sk.isConnectable() == true  
Si l'événement est OP\_CONNECT
- sk.isReadable() == true  
Si l'événement est OP\_READ
- sk.isWritable() == true  
Si l'événement est OP\_WRITE

sk.channel() : pour récupérer le channel associé

sk.attachment() : pour récupérer l'attachement.

# Selector : Concurrency

Ajout de channels entre deux appels à `Select()`

- Plusieurs threads peuvent partager le même objet `Selector`.
- Si le selector est bloqué dans un appel à `Select()`, comment ajouter un nouveau channel ?
- Pour arrêter prématurément l'appel `Select()` :
  - `Selector.Wakeup()` : Réveiller le selector
  - `Selector.Close()` : fermer le selector

# Serveur Monothread

Si on a un seul fil d'exécution, le processus doit

- attendre de nouvelles connexions
- attendre les opérations de lecture / écriture pour les clients déjà connectés
- fermer les connexions pour tous les clients qui se terminent

On peut utiliser un seul objet Selector pour toutes ces tâches.

## Exemple : Serveur Monothread

```
// Enregistrer le serveur pour accepter nouvelles connexions
server.register(selector, SelectionKey.OP_ACCEPT);
while (true) {
    selector.select();
    // Pour chaque key sélectionnée ...
    if (key.isAcceptable()) {
        SocketChannel client = serverSocket.accept();
        // Enregistrer le nouvel client
        client.register(selector, SelectionKey.OP_READ);
    }
    if (key.isReadable()) {
        SocketChannel client = (SocketChannel) key.channel();
        // Traitement de données du client ...
    }
    if (key.isWritable()) {
        SocketChannel client = (SocketChannel) key.channel();
        // Traitement de données du client ...
    }
}
```

## Serveur MultiThread avec Select()

C'est plus facile d'utiliser plusieurs threads pour gérer plusieurs connexions.

- Le processus principal attend des nouvelles connexions
- quand un nouveau client arrive, le processus peut
  - créer un nouveau thread pour gérer cette connexion
  - utiliser un thread existant pour gérer cette connexion
- Les threads peuvent utiliser le même Selector ;
- Un seul thread doit être bloqué sur l'appel Select() ; les autres threads s'exécutent en parallèle.



# Threads et Concurrency

# Processus

Un **processus** est un **programme** (statique) en cours d'**exécution** (dynamique):

Programme + environnement :

- Espace d'adressage
- pc: program counter, pile d'appels
- Objets d'entrée/sortie (entrée/sortie standard, socket, etc.)

Plusieurs processus s'exécutent simultanément :

- Système d'exploitation multi-tâche
- Le système d'exploitation alloue des **ressources** aux processus:
  - Temps processeur, mémoire, entrée/sortie
- Exécution à tour de rôle sur un seul core, illusion du parallélisme

## Threads (processus légers)

- Un **thread** est un fil d'exécution dans un programme, qui est lui même exécuté par un processus
- Un processus peut avoir plusieurs threads
  - Il est alors multi-threadé
  - Au minimum il y a un thread (`main`)
- Chaque fil d'exécution est distinct des autres et a pour attributs
  - Un point courant d'exécution (**pointeur d'instruction** ou **PC (Program Counter)**)
  - Une pile d'exécution (**stack**)
- Un thread partage tout le reste de l'environnement avec les autres threads du même processus
- La JVM est **multi-threadée** et offre au programmeur la possibilité de manipuler des threads
- Il n'est pas précisé comment ces threads sont pris en charge par le système d'exploitation

# Threads en Java

- **Interface Runnable**
  - une seule méthode `void run()`
- **Class Thread**
  - méthode `void run()`
    - Par défaut ne fait rien
    - Implémentation fournie par les extensions de Thread
  - + nombreuses autres de manipulation des threads

## Threads : programme

- Le programme (partie statique) est l'implémentation de la méthode run()

```
Class MyRunnable implements Runnable{  
    public void run(){  
        System.out.println("hello");  
    }  
}
```

OU

```
Class MyThread extends Thread{  
    public void run(){  
        System.out.println("hello");  
    }  
}
```

## Thread : exécution

Lancer l'exécution d'un thread : méthode `start()`

```
Class MyRunnable implements Runnable{  
    public void run(){  
        System.out.println("hello");  
    }  
  
    public static void main(String args[]){  
        Runnable myRunnable = new MyRunnable();  
        Thread t = new Thread(myRunnable);  
        t.start();  
    }  
}
```

## Thread : exécution

Lancer l'exécution d'un thread : méthode `start()`

```
Class MyThread extends Thread{  
    public void run(){  
        System.out.println("hello");  
    }  
  
    public static void main(String args[]){  
        Thread t = new MyThread();  
        t.start();  
    }  
}
```

## Thread et Runnable

- A chaque thread est associé une instance d'une implémentation de **Runnable**
  - cf. Constructeur public `Thread(Runnable target)`
- Le code de la méthode `run` est exécuté, le thread termine lorsque cette méthode termine
- un même objet implémentant `Runnable` peut être associé à plusieurs threads
  - chaque thread exécute de façon concurrente la méthode `run()` de l'objet passé au constructeur



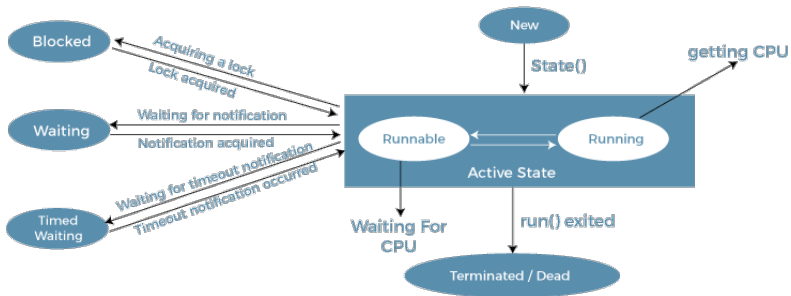
# Classe java.lang.Thread

Plusieurs attributs :

- `String name` : nom du thread
- `long id` : ID du thread
- `Thread.State state` : état
  - `NEW`, `Runnable`, `Blocked`, `Waiting`, etc.
- etc.

Accesseurs dans `java.lang.Thread`

# Etats d'un thread



Life Cycle of a Thread

# Création et contrôle des threads

- Création
  - `Thread(Runnable target) Thread(Runnable target, String name)`
- Contrôle de l'exécution d'un thread
  - `void start()`
    - démarre un thread
    - appel de la méthode `run()` du `Runnable` associé
  - `void join()` Attente de la fin d'un thread
  - `void interrupt()`
    - statut du thread positionné à `INTERRUPTED`
    - sans effet immédiat (mais permet au thread de savoir qu'un autre thread souhaite l'interrompre)
- pas de méthode pour arrêter/terminer un thread

# Méthodes statiques de la classe Thread

- `Thread.currentThread()`  
récupère l'objet Thread courant  
utile pour obtenir par exemple l'identifiant
- `boolean isInterrupted`  
est-ce que le thread courant a reçu une demande d'interruption
- `void sleep(long ms)`  
Faire dormir le thread pour une durée exprimée en ms
- `void yield()`  
Le thread renonce temporairement à la suite de son exécution

## Exemple : Serveur multi-thread

- Serveur envoi "Hello!" à chaque client
- A chaque connexion, nouveau thread en charge de la communication

## Serveur multi-thread

```
public class HelloWorker implements Runnable{
    private Socket socket;

    public HelloWorker(Socket socket){
        this.socket = socket;
    }

    public void run(){
        try(PrintWriter writer =
            new PrintWriter(socket.getOutputStream())){
            writer.write("Hello!\n");
            writer.flush();
            socket.close();
        }
        catch(IOException e){
            System.err.println("IO Error:  writing to socket");
        }
    }
}
```

## Serveur multi-thread

```
public static void main(String[] args){  
    try(ServerSocket ss = new ServerSocket(1234)){  
        while(true){  
            (new Thread(new HelloWorker(ss.accept()))).start();  
        }  
    }  
    catch(IOException e){  
        System.err.println("IO error");  
    }  
}
```

# Problèmes de la concurrence

```
public class Counter{  
    private int value = 0;  
  
    public void increment(){  
        this.value++;  
    }  
  
    public int getValue(){  
        return this.value;  
    }  
}
```



## Un compteur partagé

```
public class CountWorker implements Runnable{

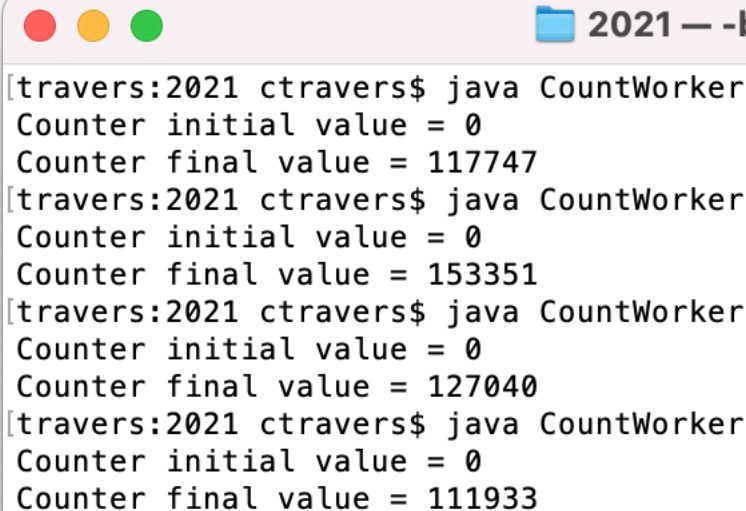
    private Counter counter;
    public CountWorker(Counter counter){
        this.counter = counter;
    }

    public void run(){
        for(int i=0; i< 10000; i++)
            counter.increment();
    }
}
```

## Un compteur partagé

```
public static void main(String[] args) throws InterruptedException {
    Thread[] threads = new Thread[20];
    Counter counter = new Counter();
    System.out.println("Counter initial value = "
                       +counter.getValue());
    for(int i=0; i< 20; i++){
        threads[i] = new Thread(new CountWorker(counter));
        threads[i].start();
    }
    for(int i=0; i< 20; i++){
        threads[i].join();
    }
    System.out.println("Counter final value = "
                       +counter.getValue());
}
```

## Compteur partagé



A terminal window with a title bar containing three colored circles (red, yellow, green) on the left and a folder icon followed by the text "2021 — -" on the right. The terminal displays four sequential runs of a Java program named "CountWorker". Each run shows the initial value of a counter (0) and the final value after execution.

```
[travers:2021 ctravers$ java CountWorker  
Counter initial value = 0  
Counter final value = 117747  
[travers:2021 ctravers$ java CountWorker  
Counter initial value = 0  
Counter final value = 153351  
[travers:2021 ctravers$ java CountWorker  
Counter initial value = 0  
Counter final value = 127040  
[travers:2021 ctravers$ java CountWorker  
Counter initial value = 0  
Counter final value = 111933
```

# Non-atomicité

`this.value++`

En fait 3 étapes :

1. `tmp = this.value`
2. `tmp = tmp + 1`
3. `this.value = tmp`

# Entrelacement

2 threads : T0 et T1

1. T0 tmp = this.value
2. T1 tmp = this.value
3. T1 tmp = tmp + 1
4. T1 this.value = tmp
5. T0 tmp = tmp + 1
6. T0 this.value = tmp

# Entrelacement

2 threads : T0 et T1

1. T0 tmp = this.value

2. T1 tmp = this.value

3. T1 tmp = tmp + 1

4. T1 this.value = tmp

5. T0 tmp = tmp + 1

6. T0 this.value = tmp

this.value = 1 !

# Programmation concurrente

- Aucune supposition sur l'entrelacement des exécutions des threads
- Tout ordre des instruction est possible

## Solutions

- Rendre un bloc d'instructions **atomique** : partie du code en **section critique**
- Utiliser des structures de données concurrentes. cf paquet **`java.util.concurrent`**
- Types Atomic (cf. Cours M1)

# Section Critique

Partie du code associé à un “verrou”

- en entrant, un thread ferme le verrou
- en sortant, le verrou est réouvert
- tant que le verrou est fermé, aucun autre thread ne peut rentrer

A tout moment, au plus un thread exécute le code de la section critique



## mot-clé Synchronized

Tout objet possède un verrou intrinsèque

- déclarer une méthode `synchronized` `public synchronized`  
`int increment()`
- ou verrouiller un bloc de code `synchronized(object){ ... /*`  
`code à verrouiller */ ... }`

Pour un objet `obj` donné, à tout un moment, au plus un thread exécute l'une des méthodes `synchronized` de l'objet, ou les blocs de code synchronisés par `synchronized(obj)`

# Attention

Ne pas synchroniser n'importe comment !!!

- les verrous sont associés à des objets
  - Deux codes synchronisés sur le même objet ne pourront pas être exécutés en même temps
- Synchroniser des parties de code qui terminent, sinon le verrou reste bloquer pour toujours
- Attention aux **deadlocks** !
  - Deux thread attendent que le verrou pris par l'autre thread se libère

Remarque :

```
synchronized int method() {... } est pareil que int  
method( ....) { synchronized(this){...}}
```

Un unique verrou associé à chaque objet

# Programmation concurrente

Conseils :

- Utilisez `synchronized` avec grande prudence !
  - Bugs faciles à créer, difficiles à détecter et corriger
  - Appeler les méthodes/blocs synchronisés dans le même ordre
- Utilisez plutôt les structures de données concurrentes de Java
  - Supportent les accès concurrent
  - Synchronisation au sein de la structure, vous n'avez pas à vous en occuper
  - Performantes : selon la structure, les opérations peuvent se faire en parallèle

# Producteurs/Consommateurs

- Les producteurs écrivent des données dans un bufffer
- Les consommateurs lisent les données écrites
- L'emplacement d'une donnée lue peut-être réutilisé
- producteur : **Ne pas écraser une donnée non lue**
- consommateur : **Ne pas lire deux fois la même donnée**

Problèmes potentiels :

- Consommateur trop rapide, peuvent lire plusieurs fois les mêmes données
- Producteur trop rapides, données risquent d'être écrasés

## ConcurrentLinkedQueue<E>

- File Thread-safe: supporte des opérations concurrentes effectuées par plusieurs thread
- Pas besoin de `synchronized` : synchronisation interne par les methode `add`, `poll` etc.
- Producteur : ajout en queue `add(E element)`
- Consommateur : récupérer la tête `poll()`
- (Implémentation lock-free : n'utilise pas de verrou)
- Limitation : opération par lot **non atomiques** (`addAll`, `removeAll`, `retainAll`, `containsAll`, `toArray...`)