

# Concurrence

---

Luc Courtrai

## **CONCURRENCE**

### **Les Threads JAVA**

**Université de Bretagne SUD  
UFR SSI - Departement MIS**



# Concurrence

---

## **Plan**

Notion de processus

Les appels système UNIX

La synchronisation entre processus

La communication entre processus

Problème d'interblocage

**Les threads JAVA**

Les Posix threads



# Concurrence : les threads

---

Sous UNIX, un processus lourd est l'exécution d'un binaire (fichier) et chaque processus possède son espace adressage (segments texte, Pile, Data), ses descripteurs de fichiers, gestionnaire de signaux, son contexte.

Le système effectue la commutation de contexte entre chaque processus (cas du temps partagé).



# Concurrence : les threads

---

## **Processus légers (ou threads):**

- [Plusieurs activités dans un processus lourd]
- Partage des ressources du processus (mémoire, (segments), [fichiers ouverts] ...)
- Efficacité du changement de contexte
- Communication rapide entre threads via la mémoire partagée



# Concurrence : les threads

---

## Processus légers :

- un compteur ordinal
- une pile d'exécution
- ses registres
- un état
- [gestionnaire de signaux]

## Processus lourds

- l'espace d'adressage (Segments)
- éventuellement:
  - *les fichiers ouverts (descripteurs)*
  - *[ gestionnaire de signaux ]*



# Concurrence : les threads

---

Mode Utilisateur (USER LEVEL)

Les threads sont applicatifs  
(non connus par le système  
d'exploitation)

API avec une bibliothèque de  
fonctions (POSIX)

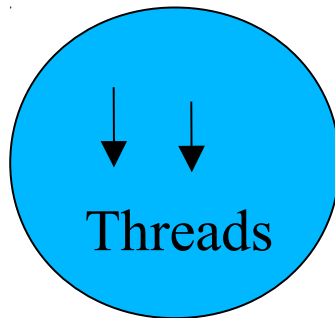
Cette bibliothèque contient  
l'ordonnanceur de threads



# Concurrence : les threads

---

Processus lourd



Noyau



A large blue rectangle representing the kernel. Inside the rectangle, the text 'Ordonnance les processus lourds' is written.

Ordonnance les processus lourds



# Concurrence : les threads

---

Inconvénients :

- Taille des piles des threads
- Les appels systèmes bloquants





# Concurrence : les threads

---

Mode Noyau :

Les systèmes comme Linux à partir  
de 2.6.xx ...

gèrent les threads au niveau du  
noyau

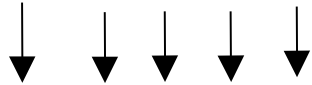
Ordonnanceur gère à la fois les  
processus lourds et les threads



# Concurrence : les threads

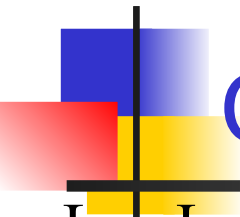
---

Processus lourds ou threads



Noyau

Ordonnance  
les processus lourds  
et threads noyaux



# Concurrence : les threads Java

---

## Les Java Threads

En Java :  
chaque activité (un thread)  
+ les objets (passifs)





# Concurrence : les threads Java

---

- La machine virtuelle Java (JVM) permet à une application d'exécuter plusieurs threads en parallèle.
- Chaque Thread a une "priorité". Un thread de haute priorité est exécuté en préférence. Par défaut un thread a la même priorité que le thread qui l'a créé.
- Les threads se partagent l'ensemble des objets de l'application. Il faut éviter les confits d'accès aux objets, attributs ou variables de classe.
- Chaque objet possède un verrou qui permettra de synchroniser les threads.
- Un thread peut être daemon (détaché (en tache de fond), non joinable) ; un thread créé par un thread daemon est aussi daemon.



# Concurrence : les threads Java

---

## Java Threads

Lorsque la machine virtuelle (JVM) démarre, il y a un thread non daemon (user) qui exécute la fonction main de l'application. La JVM continue l'exécution des threads en cours jusqu'à :

- l'appel de la méthode exit de la Classe Runtime (ou System)
- tous les threads non daemon soient arrêtés (morts)
- propagation d'une exception sans try ... catch.



# Concurrence : les threads Java

---

Java Threads : création d'un thread

Hériter de la classe **Thread** et surcharger la méthode **run**

```
class PrimeThread extends Thread {  
  
    public void run() {  
        // compute primes larger than minPrime  
        ...  
    }  
}
```



# Concurrence : les threads Java

---

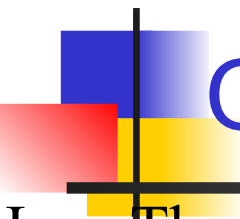
Java Threads : création d'un thread

Création (instanciation) et lancement

```
PrimeThread p = new PrimeThread();  
p.start();
```

On a ici deux activités, le thread courant ex main + le nouveau thread p

start : la JVM lance le thread qui exécute le méthode run



# Concurrence : les threads Java

---

Java Threads : la classe Thread :

**public static void sleep(long millis[,int nanos]) throws InterruptedException**

endort le threads pendant un nombre spécifié de millisecondes.





# Concurrence : les threads Java

---

Java Threads : la classe Thread :

```
public final void setPriority(int newPriority)  
public final int getPriority()
```

Change la priorité d'un thread entre MAX\_PRIORITY,  
MIN\_PRIORITY

en fonction de l'implantation de la JVM



# Concurrence : les threads Java

---

Java Threads : la classe Thread :

**public final void join([long millis [int nanos]]) throws  
InterruptedException**

attend la fin du thread

Un timeout (delai) peut être spécifié. Si le thread n'est pas terminé durant le délai, le thread appelant est débloqué.



# Concurrence : les threads Java

---

Java Threads : la classe Thread :

**public final void yield()**

passse le CPU au thread suivant



# Concurrence : les threads Java

---

Java Threads : la classe Thread :

```
public final boolean isDaemon()  
public final void setDaemon(boolean on)
```

Positionne et teste le thread comme daemon ou user (demon: non joinnable).

La méthode peut être appelée avant l'appel de la méthode start.

La JVM s'arrête lorsqu'il n'y a plus que des threads daemons.  
Le thread main est user



# Concurrence : les threads Java

---

Java Threads : Runnable

## **AUTRE FACON DE CRÉER DES THREADS**

L'interface Runnable

- permet hériter d'une autre classe
- dissocier le thread (controleur) de l'action



# Concurrence : les threads Java

---

Java Threads : création d'un thread avec runnable

```
class PrimeRun implements Runnable {  
    public void run() {  
        // compute primes larger than minPrime  
        ...  
    }  
}  
...  
PrimeRun p = new PrimeRun();  
Thread th=new Thread(p)  
th.start(); /// sur le thread et non sur le runnable  
th.join();
```

On utilise un thread qui gère le runnable

Il faut passer un runnable au constructeur de la classe Thread



# Concurrence : les threads Java

---

## **Runnable + son thread attaché (même utilisation que extends Thread)**

```
public class PrinRun implements Runnable {
    protected Thread unThread;

    public PrinRun( ) {
        unThread = new Thread((Runnable)this);
    }
    public void start() { // pour avoir la même interface qu'un thread
        unThread.start();
    }
    public void join() throws InterruptedException {
        unThread.join();
    }
    public void run() {
        // compute primes larger than minPrime
        .....
    }
}
```



# Concurrence : les threads Java

---

Java Threads : création d'un thread      Runnable + son thread attaché

```
PrimeRun p = new PrimeRun();
```

```
p.start();
```

```
p.join()...
```

```
....
```





# Concurrence : les threads Java

---

## Java Threads : **Groupe de threads**

Chaque thread appartient à un groupe de threads

- Structurer les applications
- Contrôler un ensemble de threads
- ....



# Concurrence : les threads Java

---

Java Threads : Groupe de threads

```
public class UseThreads {
```

```
    public static void main(String args[]) {
```

```
        ThreadGroup tGr = new ThreadGroup("Groupe de Threads ");
```

```
        Slave a = new Slave(tGr,"threadA"); // un thread
```

```
                                           // le gid + nom symbolique du thread
```

```
        a.start();
```

```
        System.out.println(tGr.activeCount());
```

```
        Slave b = new Slave(tGr,"threadB");
```

```
        b.start();
```

```
        System.out.println("nombre de threads "+ tGr.activeCount());
```

```
        Thread[] list = new Thread[] ;
```

```
        tGr.enumerate(list);
```

```
        for (Thread th:list) System.out.println(th) ;
```



# Concurrence : les threads Java

---

## Java Threads : **Synchronisation**

Tout Objet Java possède un verrou .

Ces verrous sont manipulés par la clause `synchronized` qui exprime une section critique sur l'objet (exclusion mutuelle de threads).

La clause est mise sur une méthode ou sur un bloc d'instruction `{ }`



# Concurrence : les threads Java

---

## Java Threads : Synchronisation

### Synchronisation sur un objet

```
class Toto {  
    int value;  
    public synchronized void setValue(int v){  
        value =v;  
    }  
    public synchronized int getValue(){  
        value =v;  
    }  
    public synchronized void incValue(int v){  
        value = getValue() + v; // appel réentrant  
    }  
}
```



# Concurrence : les threads Java

---

Java Threads : Synchronisation

```
Toto unToto = new Toto();  
Toto unAutreToto = new Toto();
```

Deux threads ne peuvent pas appeler une des méthodes synchronized sur l'objet "unToto".

Le deuxième appel est mis dans une file d'attente

Un thread peut appeler une des méthodes synchronized sur "unToto" pendant qu'un deuxième thread appelle une des méthodes synchronized de "unAutreToto" (deux objets -> deux verrous)



# Concurrence : les threads Java

---

## Java Threads : Synchronisation

Le Synchronized dans un bloc, il faut prendre un verrou (dans l'exemple suivant celui de l'objet)

```
class Titi
static Object o = new Object();
void m(){
    ...
    synchronized(o){// verrou de o
        // partie de code en SC
    }
    ...
}
```

Ici l'objet o peut être partagé par de objets différents (voir de classes différentes). O sert à nommer la synchro



# Concurrence : les threads Java

---

Java Threads : Synchronisation

## **verrou sur l'objet**

```
void m(){  
    ... // non protégé  
    synchronized(this){// verrou sur l'objet courant  
        // partie de code en section critique  
    }  
    ... // non protégé  
}
```



# Concurrence : les threads Java

---

Java Threads : Synchronisation  
**Verrou de l'objet Classe**

```
class Th extends Thread{
    static int Cpt =0;
    static synchronized int Incremente(){
        return Cpt++;
    }
    public Th(){
    }
    public void run(){
        while(true){
            System.out.println(this + "CPT: " +Incremente());
        }
    }
}
```





# Concurrence : les threads Java

## Java Threads : Synchronisation

```
public class Test {  
    public static void main(String args[]) {  
        Th t1 = new Th();  
        Th t2 = new Th();  
        Th t3 = new Th();  
        Th t4 = new Th();  
  
        t1.start(); t2.start(); t3.start();t4.start();  
  
        try {  
            t1.join();
```

....

t1,t2,t3,t4 sont des objets différents mais synchronisés sur l'objet classe Th



# Concurrence : les threads Java

---

## Java Threads : Synchronisation

La méthode thread

```
public static boolean holdsLock(Object obj)
```

retourne vrai si le thread courant a le verrou sur l'objet obj



# Concurrence : les threads Java

Java Threads : Synchronisation

## Wait et Notify

Abstraction de Synchronisation les Moniteurs

Synchronisation bloquante

```
public final void wait([long timeout,int nanos])  
    throws InterruptedException
```

La méthode bloque le thread courant jusqu'à ce qu'un autre thread le notifie

La synchronisation est nommée par l'objet sur lequel s'applique le wait



# Concurrence : les threads Java

---

Java Threads : Synchronisation Wait et Notify

```
public final void notify()  
public final void notifyAll()
```

La méthode `notify` réveille un thread ayant effectué un `wait` sur l'objet courant. Si plusieurs threads sont bloqués sur cet objet, le choix du thread est aléatoire.

La méthode `notifyAll` réveille tous les threads bloqués sur l'objet.



# Concurrence : les threads Java

---

Java Threads : Synchronisation Wait et Notify

Pour appeler la méthode Wait ou Notify, il faut posséder le verrou de l'objet sur lequel s'applique la méthode.

Cela garantit l'exclusion mutuelle aux appels des méthodes Wait ou Notify.

Exemple : cela interdit de rentrer dans un notify si un autre thread est entrain d'effectuer un wait (sinon la programmation concurrence serait impossible).

Pour obtenir le verrou, il faut être dans un bloc synchronized.



# Concurrence : les threads Java

---

Java Threads : Synchronisation Wait et Notify

Lorsqu'un thread T1 se bloque sur un Wait, il relache automatiquement le verrou. Cela permet à un autre thread T2 d'effectuer le Notify.

Après ce Notify, le premier thread T1 attend qu'il réoptienne le verrou pour continuer son exécution. Il rentre alors en compétition avec les autres threads qui veulent obtenir aussi le verrou (synchronized).



# Concurrence : les threads Java

Java Threads : Wait et Notify

thread-1

```
synchronized(o) {  
  
    ....  
    try {  
        o.wait();  
    ...  
    } catch(..) {..  
  
    ...  
}
```

thread-2

```
synchronized(o) {  
  
    ....  
    o.notify();  
    ...  
}
```



# Concurrence : les threads Java

Java Threads : Wait et Notify

thread-t1

```
synchronized(o) {  
    prend le verrou o  
    try {  
        o.wait(); bloque t1  
        ...  
        relache le verrou de o  
    }  
}
```

thread-t2

```
synchronized(o) {  
    prend le verrou o  
    o.notify(); debloque t1  
    ...  
} relache le verrou
```

**attend le verrou**

**prend le verrou**

```
} catch(..) {..}...
```

**relache le verrou**





# Concurrence : les threads Java

Java Threads : Wait et Notify Exemple :synchro

Exemple de synchro (un ou n thread(s) doivent attendre le fin d'une tache)

```
class SynchroL {
```

```
    Object lock; // un Objet pour sont verrou (ou l'object this)
```

```
    boolean isLock; // le verrou pris ou pas
```

```
    public SynchroL() { // constructeur
```

```
        lock = new Object();
```

```
        isLock = True;
```

```
    }
```



# Concurrence : les threads Java

Java Threads : Wait et Notify Exemple :synchro

```
public void testSynchro(){ // bloquante
    synchronized (lock){
        if (isLock) //il faut le verrou de l'objet pour le wait
            try {
                lock.wait(); // libère aussi le verrou
            } catch (InterruptedException e) {}
    }
}

public void ok(){
    synchronized(lock){ // prend le verrou pour le notify
        lock.notifyAll(); // libère le wait à la fin de ce bloc synchronized
        isLock = false;
    }
}
```



# Concurrence : les threads Java

Java Threads : Wait et Notify Exemple : synchro

```
class T1 extends Thread{
    SynchroL unLock;
    public T1(SynchroL loc){
        this.unLock = loc;
    }
    public void run(){
        ....
        System.out.println("Avant Synchro");
        unLock.testSysnchro();
        System.out.println("Apres Synchro");
        //...
    }
}
```



# Concurrence : les threads Java

Java Threads : Wait et Notify Exemple : synchro

```
class T2 extends Thread {  
    SynchroL unLock;  
    public T2(SynchroL l){  
        this.unLock = l;  
    }  
    public void run(){  
        // Etape1;  
        System.out.println("Avant ok");  
        unLock.ok(); // previent de la fin de t1  
        System.out.println("Apres ok");  
  
        //...  
    }  
}
```



# Concurrence : les threads Java

---

Java Threads : Wait et Notify Exemple : synchro

```
public class Test {  
    public static void main(String args[]) {  
        SynchroL sync = new SynchroL();  
        T1 t1 = new T1(sync);  
        T2 t2 = new T2(sync);  
        t1.start();  
        t2.start();  
        ...  
    }  
}
```



# Concurrence : les threads Java

Java Threads : Wait et Notify Exemple : **Producteur-consommateur**

```
class TamponCirc {  
  
    private Object tampon[];  
    private int taille;  
    private int in, out, nbMess;  
  
    // constructeur d'un tampon borne  
    public TamponCirc (int taille) {  
        tampon = new Object[taille]; // ou Vector  
        this.taille = taille;  
        in = 0;  
        out = 0;  
        nbMess = 0;  
    }  
}
```



# Concurrence : les threads Java

Java Threads : Wait et Notify Exemple : producteur-consommateur

```
public synchronized void depose(Object obj) {  
    while (nMess == taille) { // si plein  
        try {  
            wait();           // attends non-plein  
        } catch (InterruptedException e) {}  
    }  
    tampon[in] = obj;  
    nbMess++;  
    in = (in + 1) % taille;  
    notify();                // envoie un signal non-vidé  
}
```



# Concurrence : les threads Java

Java Threads : Wait et Notify Exemple : producteur-consommateur

```
public synchronized Object preleve() {  
    while (nMess == 0) {    // si vide  
        try {  
            wait();        // attends non-vide  
        } catch (InterruptedException e) {}  
    }  
    Object obj = tampon[out];  
    tampon[out] = null;    // supprime la ref a l'objet pour le GC  
    nbMess--;  
    out = (out + 1) % taille;  
    notify();              // envoie un signal non-plein  
    return obj;  
}
```





# Concurrence : les threads Java

Java Threads : Wait et Notify Exemple : producteur-consommateur

```
class Consommateur extends Thread {  
  
    private TamponCirc tampon;  
    public consommateur(TamponCirc tampon) {  
        this.tampon = tampon;  
    }  
    public void run() {  
        while (true) {  
            System.out.println("je preleve "+((Integer)tampon.preleve()).toString());  
            try {  
                Thread.sleep((int)(Math.random()*200)); // attends jusqu'a 200 ms  
            } catch (InterruptedException e) {}  
        }  
    }  
}
```



# Concurrence : les threads Java

Java Threads : Wait et Notify Exemple : producteur-consommateur

```
class producteur extends Thread {  
    private tamponCirc tampon;  
    private int val = 0;  
    public producteur(tamponCirc tampon) {  
        this.tampon = tampon;  
    }  
    public void run() {  
        while (true) {  
            System.out.println("je depose "+val);  
            tampon.depose(new Integer(val++));  
            try {  
                Thread.sleep((int)(Math.random()*100)); // attend jusqu'a 100 ms  
            } catch (InterruptedException e) {}  
        }  
    }  
}
```



# Concurrence : les threads Java

Java Threads : Wait et Notify Exemple : producteur-consommateur

```
class TestTampon {  
  
    public static void main(String args[]) {  
  
        tamponCirc tampon = new tamponCirc(5);  
        producteur prod = new producteur(tampon);  
        consommateur cons = new consommateur(tampon);  
  
        prod.start();  
        cons.start();  
  
    }  
  
}
```



# Concurrence : les threads Java

---

Tampon non borné, ordre FIFO des retraits.

Implantation d'une file bloquante

package java.util.concurrent

## LinkedBlockingQueue<E>

```
public LinkedBlockingQueue<E>() ...
```

```
// crée une BlockingQueue avec une liste de noeuds. Les  
accès aux méthodes par les threads respectent l'ordre FIFO  
des demandes.
```

```
public void put(<E> o) ;
```

```
// insère l'élément o à la fin de la file (pas limite de taille)
```

```
public <E> take() ;
```

```
// retourne et supprime la tête de la file, attend si aucun  
élément n'est présent dans la file.
```



# Concurrence : les threads Java

---

```
public class LinkedBlockingQueue<E>{  
    // BlockingThread classe interne pour associer un thread  
    bloqué en attente d'un objet et une référence sur l'objet  
    class BlockingThread{  
        private Thread thread; // reference du thread bloqué  
        private E o; // référence pour l'objet à récupérer  
        public BlockingThread(Thread th){  
            this.th=th;  
        }  
        public void setObject(E o){ this.o=o;}  
        public E getObject(){return o;}  
        public Thread getThread(){return thread;}  
    }  
}
```



# Concurrence : les threads Java

---

```
public class LinkedBlockingQueue<E> {  
    private LinkedList<BlockingThread> listeTh;  
    private LinkedList<E> listeObject;
```

```
    public LinkedBlockingQueue()  
    {  
        listeTh = new LinkedList<BlockingThread>();  
        listeObject = new LinkedList<E>();  
    }
```



# Concurrence : les threads Java

---

```
public class LinkedBlockingQueue<E>{
```

```
    public synchronized void put(E o){
```

```
        if (listeTh.size() == 0){
```

```
            listeObject.add(o);
```

```
        } else
```

```
            BlockingThread tb = listeTh.remove();
```

```
            tb.setObject(o);
```

```
            synchronized(tb.getThread()){
```

```
                tb.getThread().notify();
```

```
            }
```

```
        }
```



# Concurrence : les threads Java

---

```
public E take() {  
    Thread my=Thread.currentThread();  
    BlockingThread tb =null;  
    synchronized(my){  
        synchronized(this){  
            if (listeObject.size() != 0)  
                return listeObject.remove();  
            tb = new BlockingThread(my); // sinon  
            listeTh.add(tb);  
        } // synchronized(this)  
        try{  
            my.wait();  
            return tb.getObject();  
        }catch(InterruptedException e){};  
    } // synchronized(my)  
}
```

```
}
```





# Concurrence : les threads Java

---

## Java Threads

### Interruptible

**Interrompt un thread bloqué dans un sleep, wait ....  
(ie du point d'annulation : endroit où un thread peut être arrêté)**

**t1**

```
synchronized (this) {  
    Try {  
        wait(); // ou sleep  
    } catch (InterruptedException e) {  
        System.out.println(this + "Exception levee + " + e)  
    }  
}
```

**t2**

```
t1.interrupt();
```



# Concurrence : les threads Java

---

## Java Threads

### Interruptible

**contrôler les interrupt()**

**t1**

```
synchronized (this) {  
    try {wait();}  
    catch (InterruptedException e) {  
        System.out.println(this + "Exception levee " + e)  
    }  
}
```

**public void interrupt(); // Redéfinir la méthode interrupt**

**t2**

**t1.interrupt();**



# Concurrence : les threads Java

---

Java Threads : `java.util.concurrent.Semaphore`

**Utiliser les algorithmes classiques :**

- **producteur-consomateur**
- **lecteur-rédacteur**




# Concurrence : les threads Java

---

Java Threads : `java.util.concurrent.Semaphore`

**Class `java.util.concurrent.semaphore`**

```
public Semaphore(int permits)
    permits peut être négatif
public Semaphore(int permits, boolean fair)
    fair // true  FIFO acquire
        // false aléatoire (NE PAS UTILISER)
```



# Concurrence : les threads Java

---

Java Threads : `java.util.concurrent.Semaphore`

*// operation P*

**public void acquire()  
    throws InterruptedException**

**public void acquireUninterruptedException()**

*// operation V*

**public void release()**



# Concurrence : les threads Java

---

Java Threads : java.util.concurrent.Semaphore

exemple le tampon borné avec les sémaphores

```
class tamponCirc {
```

```
    private Object tampon[];
```

```
    private Semaphore sMutex,sPlaceLibre,sArticle;
```

```
    private int en;
```

```
    public tamponCirc (int taille) {
```

```
        tampon = new Object[taille];
```

```
        this.taille = taille;
```

```
        sMutex = new Semaphore(1,true);
```

```
        sPlaceLibre = new Semaphore(taille,true);// true FIFO
```

```
        sArticle = new Semaphore(0,true); /
```

```
        en = 0;
```

```
    }
```



# Concurrence : les threads Java

---

**Java Threads : java.util.concurrent.Semaphore**  
exemple tampon borne

```
public void depose(Object obj) {  
    sPlaceLibre.acquireUninterruptibly();  
    sMutex.acquireUninterruptibly();  
    tampon[en] = obj;  
    en = (en + 1) % tampon.size;  
    sMutex.release();  
    sArticle.release();  
  
}
```



# Concurrence : les threads Java

---

**Java Threads : `java.util.concurrent.Semaphore`**  
exemple tampon borne

```
public Object preleve() {  
    sArticle.acquireUninterruptibly();  
    sMutex.acquireUninterruptibly();  
    Object obj = tampon[hors];  
    tampon[hors] = null;    // supprime la ref a l'objet  
    hors = (hors + 1) % tampon.length;  
    sMutex.release();  
    sPlaceLibre.release();  
    return obj;  
}
```





# Concurrence : les threads Java

---

Java Threads : `java.util.concurrent.Semaphore`

Quelques autres méthodes :

```
public void acquire(int permits)
```

```
public void release(int permits)
```

```
public boolean tryAcquire(int permits)
```

```
public boolean tryAcquire(int permits,  
                           long timeout,  
                           TimeUnit unit)
```

```
    // TimeUnit.SECONDS
```

```
    // TimeUnit.MILLISECONDS
```

```
    // TimeUnit.MICROSECONDS
```

```
    // TimeUnit.NANOSECONDS
```

```
throws InterruptedException
```



# Concurrence : les threads Java

---

**Java Threads : `java.util.concurrent`.**

Les Verrous ( ie `TestAndSet`)

```
verrou = new AtomicBoolean(false)
```

dans le code concurrent

```
boolean lock= verrou.getAndSet(true);  
if ( ! lock) {  
    // en exclusion mutuelle  
    ....  
    // fin exclusion mututelle  
    verrou.set(false);  
}
```



# Concurrence : les threads Java

---

**Java Threads : `java.util.concurrent`.**

Java. concurrent.\*

Producteur consommateur

LinkedBlockingQueue

Utilisant l'algo producteur consommateur

Producteur : `void put(E elt)`

Consommateur : `E take()`

L'appel de la méthode `E take()` peut être bloquant.

Avec un ordre FIFO sur les threads bloqués sur le `take()`



# Concurrence : les threads Java

---

java.util.concurrent.CyclicBarrier

```
Class Test extends Thread {  
    static CyclicBarrier barrier;
```

```
    public void run() {  
        .....  
        try { barrier.await();  
                } catch (InterruptedException ex) {  
                    return;  
                } catch (BrokenBarrierException ex) {  
                    return;  
                }  
    }  
}
```

```
...  
barrier = new CyclicBarrier(N);  
for (int i = 0; i < N; ++i)  
    new Test().start();
```



# Concurrence : les threads Java

---

Java Threads :

Implantation des Sémaphores en JAVA au dessus des moniteurs wait/notify



# Concurrence : les threads Java

---

java.util.concurrent.CyclicBarrier

```
Class Test extends Thread {  
    static CyclicBarrier barrier;
```

```
    public void run() {  
        .....  
        try { barrier.await();  
                } catch (InterruptedException ex) {  
                    return;  
                } catch (BrokenBarrierException ex) {  
                    return;  
                }  
    }  
}
```

```
...  
barrier = new CyclicBarrier(N);  
for (int i = 0; i < N; ++i)  
    new Test().start();
```



# Concurrence : les threads Java

---

Java Threads : SpollThread Callable Futur

```
// Exemple Oracle
// les taches à effectuer
public class WordLengthCallable
    implements Callable {
    // Callable (ie Runnable) avec return
    private String word;
    public WordLengthCallable(String word) {
        this.word = word;
    }
    public Integer call() {
        return Integer.valueOf(word.length());
    }
}
```



# Concurrence : les threads Java

Java Threads : SpoolThread, Callable, Futur

// Future (synchro à l'appel de get)

```
public class TestPollThreadCallable {  
    public static void main(String args[] ) throws Exception {  
  
        String mots[] = {"0", "12", "345", "6789"};  
        ExecutorService pool = Executors.newFixedThreadPool(2);  
        Set<Future<Integer>> set = new HashSet<Future<Integer>>();  
        for (String word: mots) {  
            Callable<Integer> callable = new  
                                     WordLengthCallable(word);  
            Future<Integer> future = pool.submit(callable);  
            set.add(future);  
        }  
        int sum = 0;  
        for (Future<Integer> future : set)  
            sum += future.get();  
        System.out.printf("The sum of lengths is %s\n", sum);  
    }  
}
```





# Concurrence : les threads Java

---

java.util.Stream (pipeline sur une liste d'objets) java8  
exemple sur les int

```
InitStream flux= IntStream.range(0,10)
flux.forEach(System.out::println);
0
1
2
3
4
5
6
7
8
9
flux.forEach(System.out::println);
// erreur le flux est déjà consommé
```



# Concurrence : les threads Java

---

java.util.Stream (pipeline sur une liste d'objets) java8  
exemple sur les int

```
IntStream.range(0,10).forEach(System.out::println);
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```



# Concurrence : les threads Java

---

java.util.Stream (pipeline sur une liste d'objets) java8

filter() lambda fonction

```
IntStream.range(0,10)
    .filter(i ->i%2==0)
    .forEach(System.out::println);
```

```
0
2
4
6
8
```

# Concurrence : les threads Java

java.util.Stream (pipeline sur une liste d'objets) java8

Traitement() sur le pipeline avec une lambda fonction

```
IntStream  
    .range(0,10)  
    .map(i -> i * i )  
    .forEach(System.out::println);
```

```
0  
1  
4  
9  
16  
25  
36  
49  
64  
81
```

# Concurrence : les threads Java

java.util.Stream (pipeline sur une liste d'objets) java8

Traitement() sur le pipeline avec une lambda fonction

```
IntStream.range(0,10).map(  
    i ->  
    { return i * i ; }  
)  
    .forEach(System.out::println);
```

0  
1  
4  
9  
16  
25  
36  
49  
64

# Concurrence : les threads Java

java.util.Stream (pipeline sur une liste d'objets) java8  
traitement en parallèle (thread)

```
IntStream.range(0,10)
          .parallel()
          .map(
            i ->
            { return i * i ; }
          )
          .forEach(System.out::println);
```

16

1

25

64

16

4

0

49

36



# Concurrence : les threads Java

---

java.util.Stream (pipeline sur une liste d'objets) java8  
Reduction de la liste (thread)

```
int sum =IntStream
    .range(0,10)
    .reduce(0 // premier appel (init)
           ,(i1,i2) -> {return i1+i2;}// autres appels
           )
System.out.println(sum) :
45
```

# Concurrence : les threads Java

java.util.Stream (pipeline sur une liste d'objets) java8  
Exemple de pipeline : somme de [0, 999]

```
sum =IntStream.range(0,10)
    .map(i -> i *10)    // 0  10 20 30 ... 90
    .parallel()    // en parallèle (threads différents)
    .map(i -> {
        // traitement en parallèle
        int som=0;
        for (int s=i ; s < i +10; s++) //sum des [i ,i +10]
            som+=s;
        return som; })
    // fusion des resultats
    .reduce(    0, // premier appel (ie init)
            (i1,i2) -> {return i1+i2;}); // les autres

// 4950    (99*100)/2
```