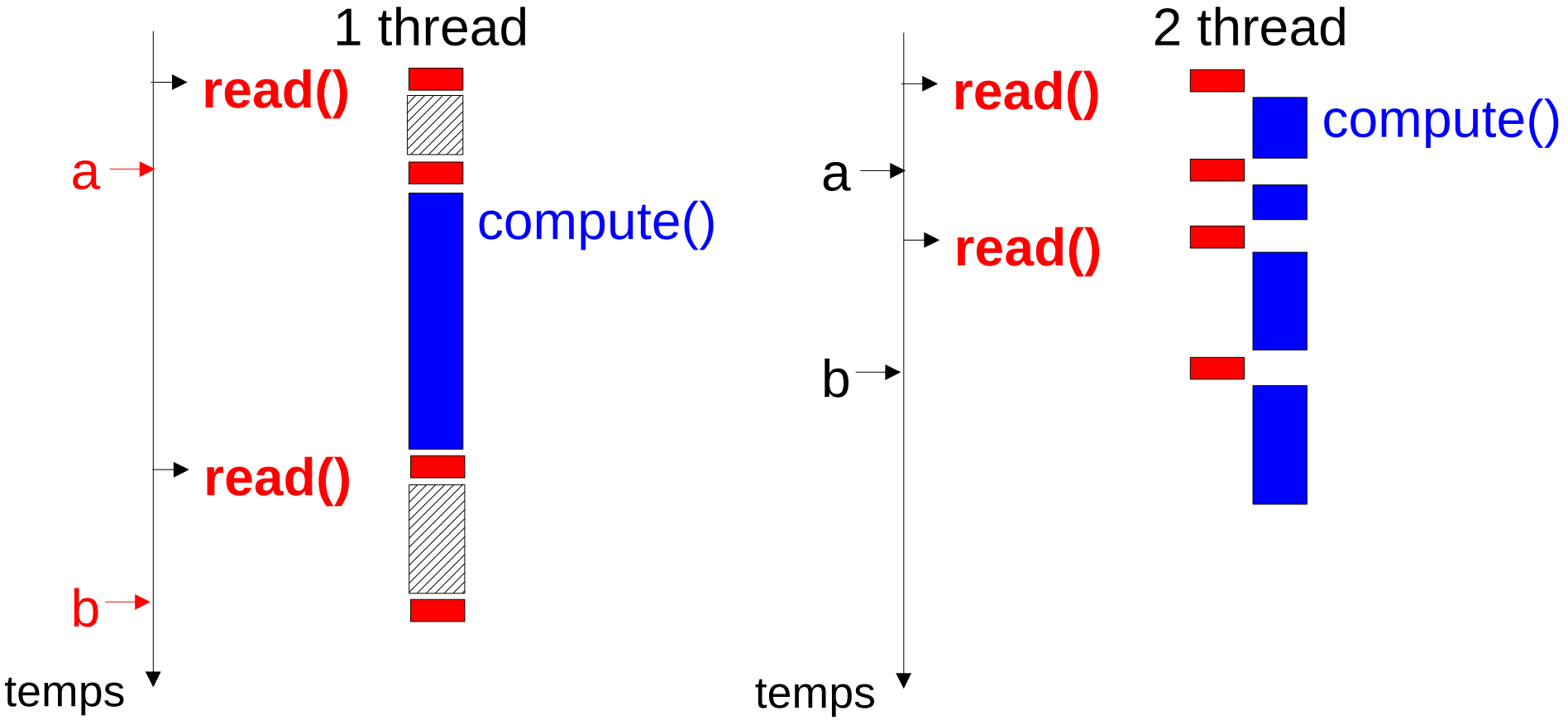
# Programmation concurrente: c'est quoi?

- Permettre d'effectuer plusieurs traitements, spécifiés distinctement les uns des autres, **en même temps**
- En général, dans la spécification d'un traitement, beaucoup de ***temps est passé à attendre***
  - Idée: exploiter ces temps d'attente pour réaliser d'autres traitements, en exécutant **en concurrence** plusieurs traitements
  - Sur mono-processeur, simulation de parallélisme
  - Peut simplifier l'écriture de certains programmes (dissocier différentes activités)

# Un exemple: calculs et interactions

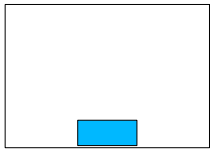
- Utilisation des temps d'attente de saisie
  - Ex (3 instructions indépendantes): **read(); compute(); read();**



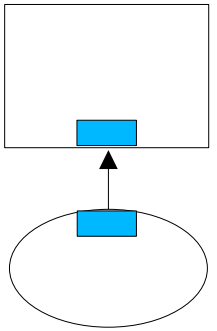
# Autre exemple: client / serveur

- L'**attente** d'un nouveau client peut se faire **en même temps** que le **service** au client déjà là.

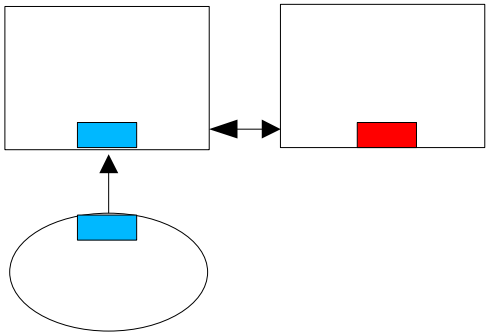
1. attente de client



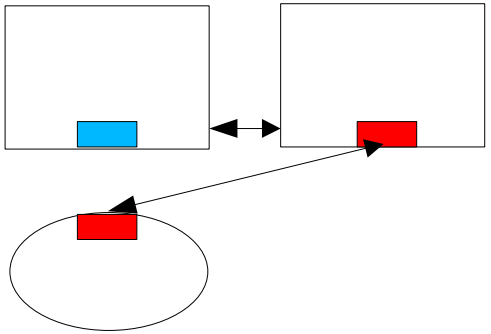
2. arrivée d'un client



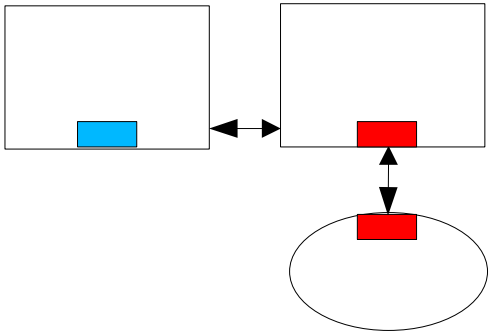
3. création d'1 thread



4. redirection de la connexion



5. attente et traitement concurrents



# Programmation concurrente en Java

- Notion de concurrence "système" (processus)
  - java.lang.Runtime, Process, ProcessBuilder
- Bien mieux géré au niveau de la machine virtuelle
  - Classes spécifiques: java.lang.Thread, Runnable...
  - Classes de service: java.util.concurrent.Executors...
  - Cohérence des valeurs (volatile et classes *atomics*), relativement au *Java Memory Model* (JMM, JSR 133)
    - Mémoire "globale" *vs* "locale" (mises à jour)
  - Protection des accès et organisation des threads, exclusion mutuelle...
    - synchronized, moniteurs, wait()/notify(), java.util.concurrent.locks, etc.

# Les Processus

- Objet représentant une application qui s'exécute
  - java.lang.Runtime
    - Objet de contrôle de l'environnement d'exécution
      Objet courant récupérable par Runtime.getRuntime()
    - D'autres méthodes: [total/free/max]Memory(), gc(), exit(), halt(), availableProcessors()...
    - exec() crée un nouveau processus
  - java.lang.Process et ProcessBuilder
    - Objets de contrôle d'un (ensemble de) processus, ou commandes
    - Runtime.getRuntime().exec("cmd") crée un nouveau processus correspondant à l'exécution de la commande, et retourne un objet de la classe Process qui le représente

# La classe Runtime

- Les différentes méthodes exec() créent un processus natif.
  - Exemple simple:

```
Runtime.getRuntime().exec("javac MonProg.java");
```

  - Avec un tableau d'arguments

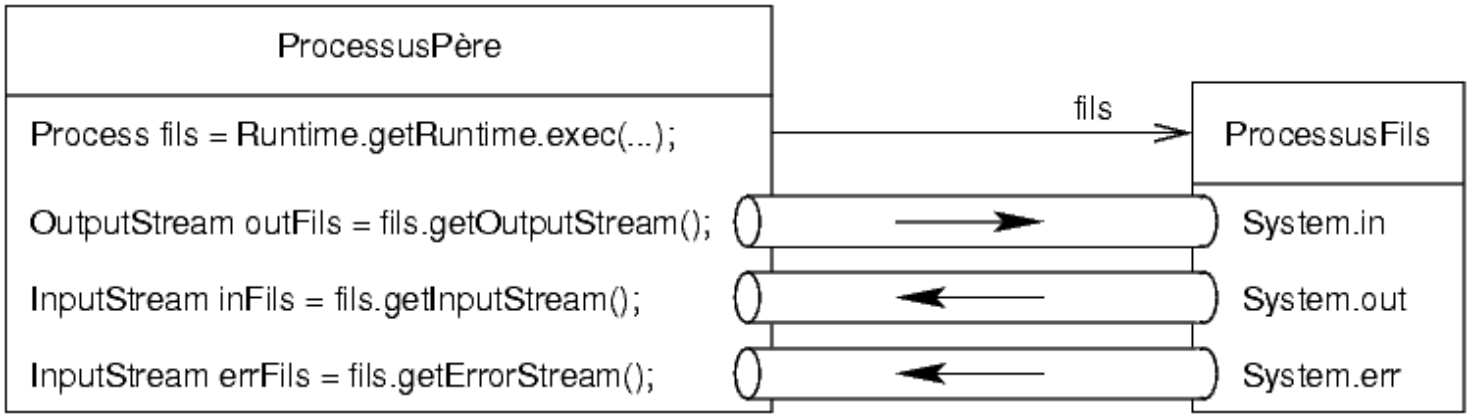
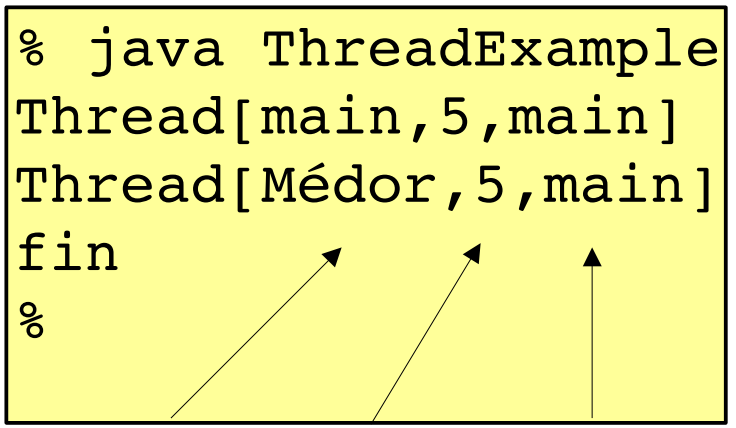
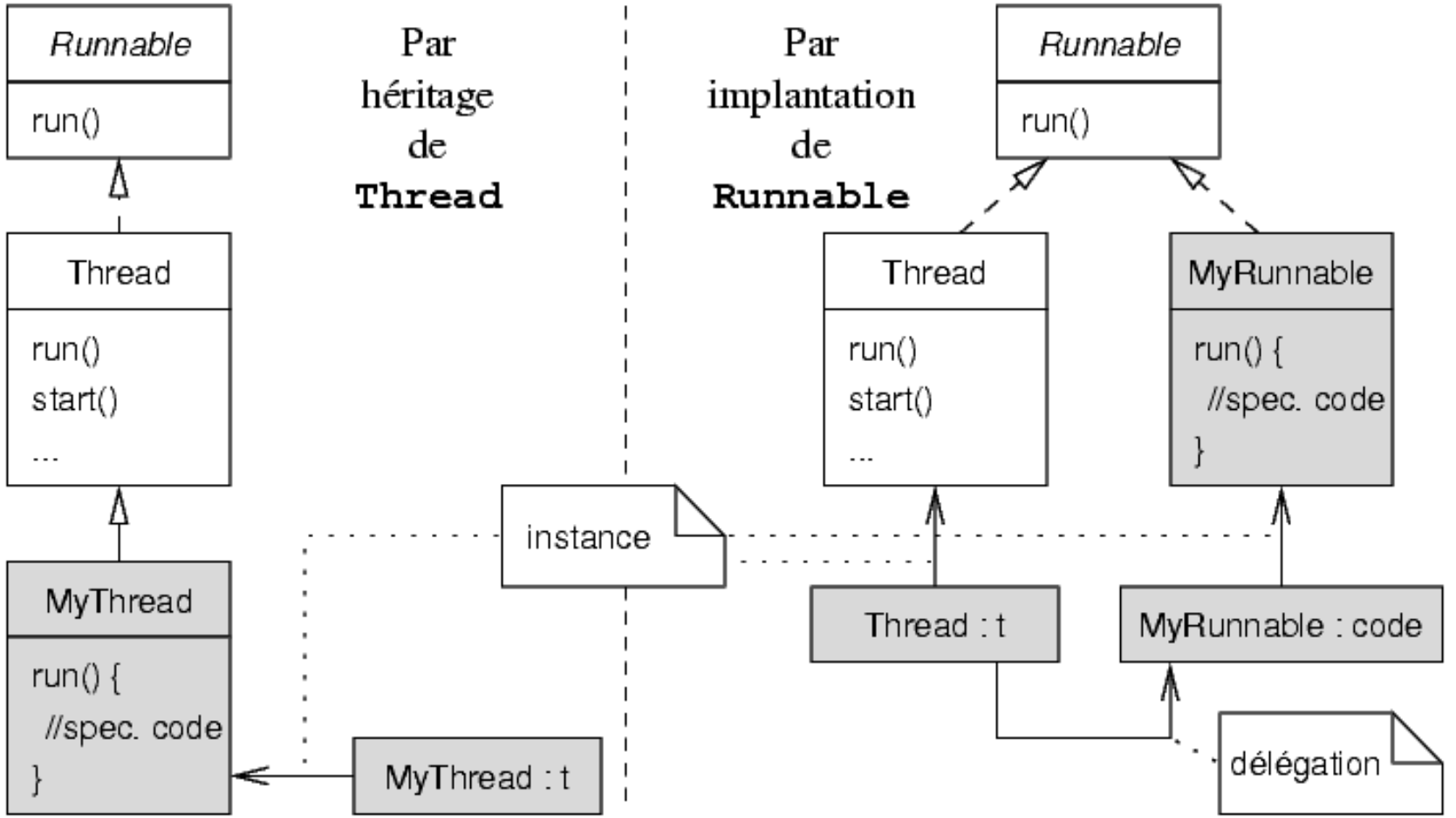
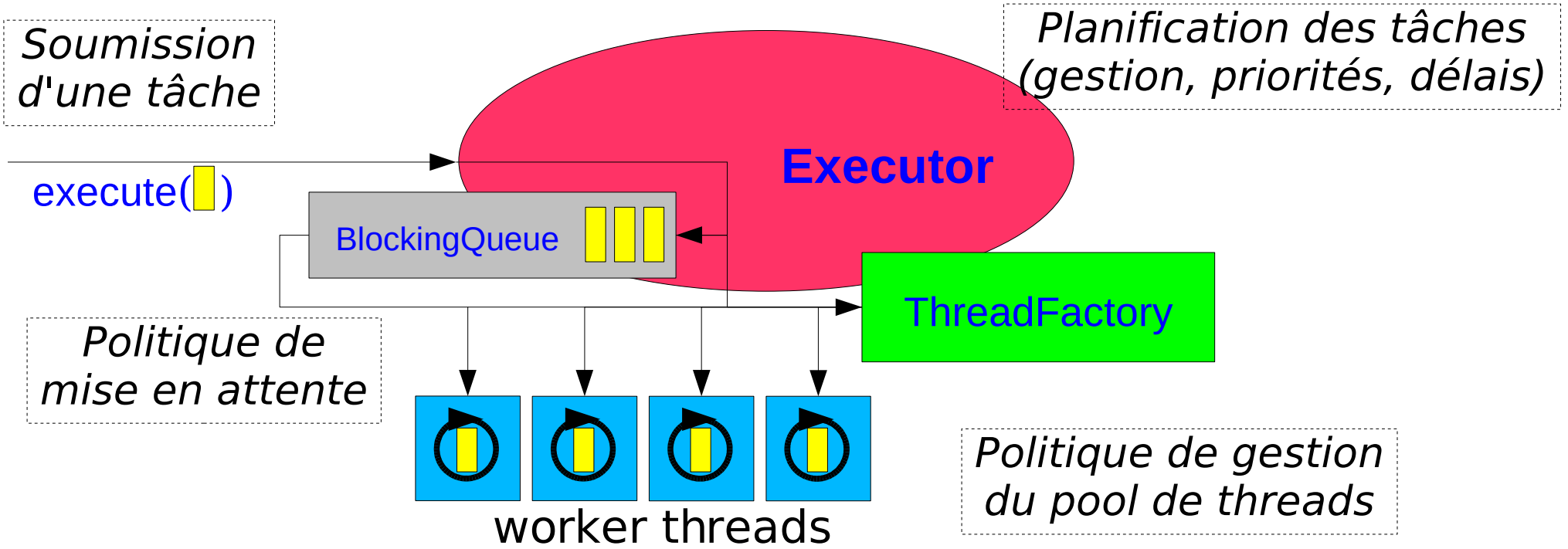
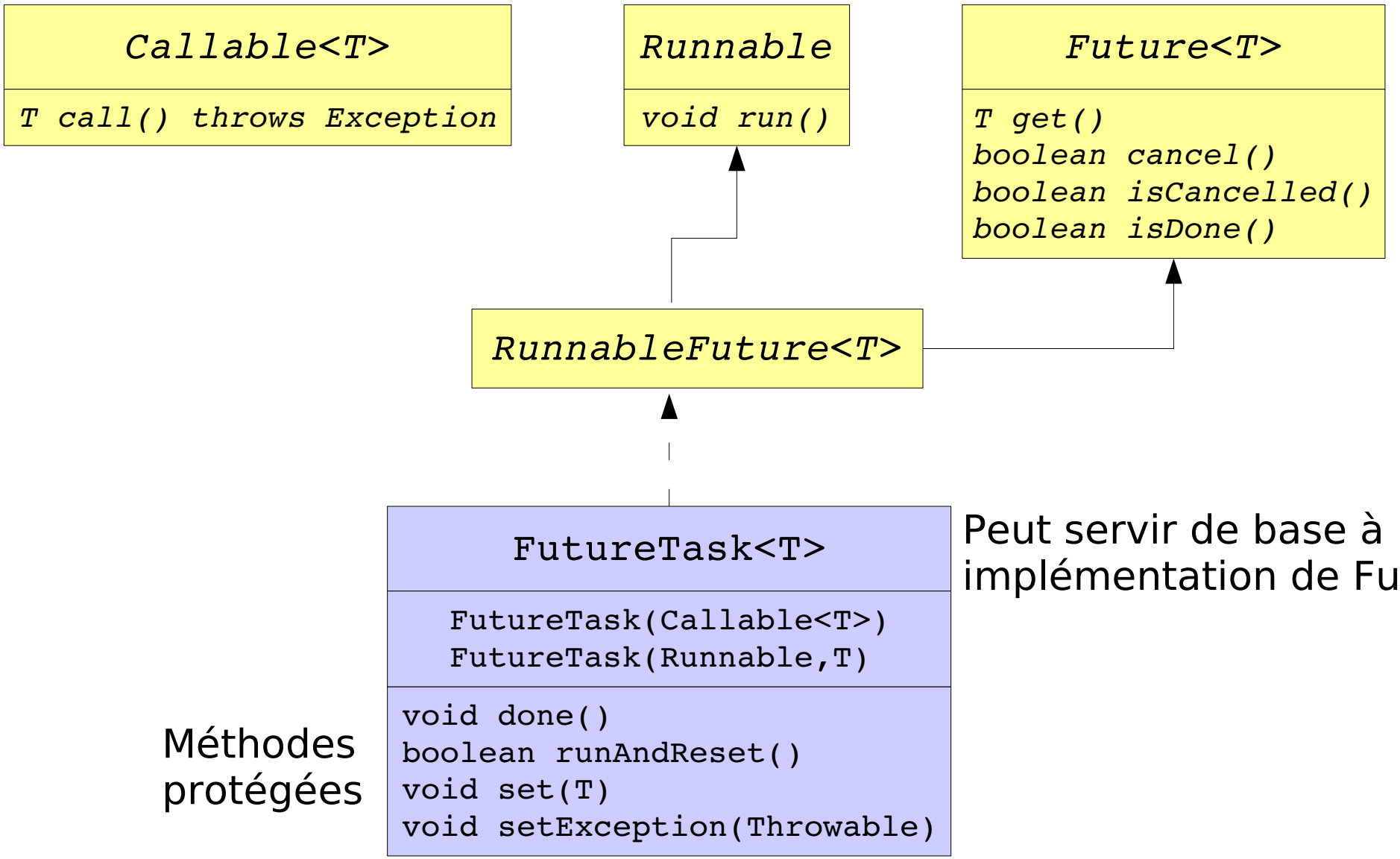
```
Runtime.getRuntime()
       .exec(new String[]{"javac", "MonProg.java"});
```

  - Exécute la commande cmd dans le répertoire /tmp avec comme variable d'environnement var la valeur val.

```
Runtime.getRuntime()
     .exec("cmd",
           new String[] {"var=val"},
           new File("/tmp/"));
```

# La classe Process

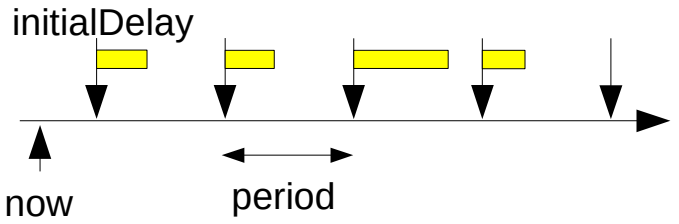
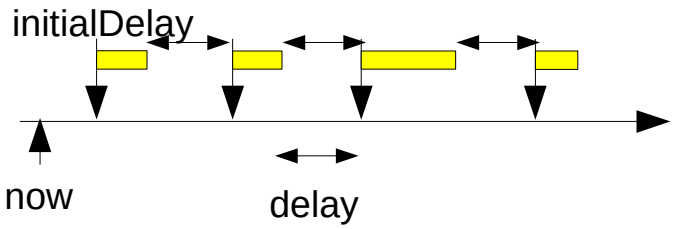
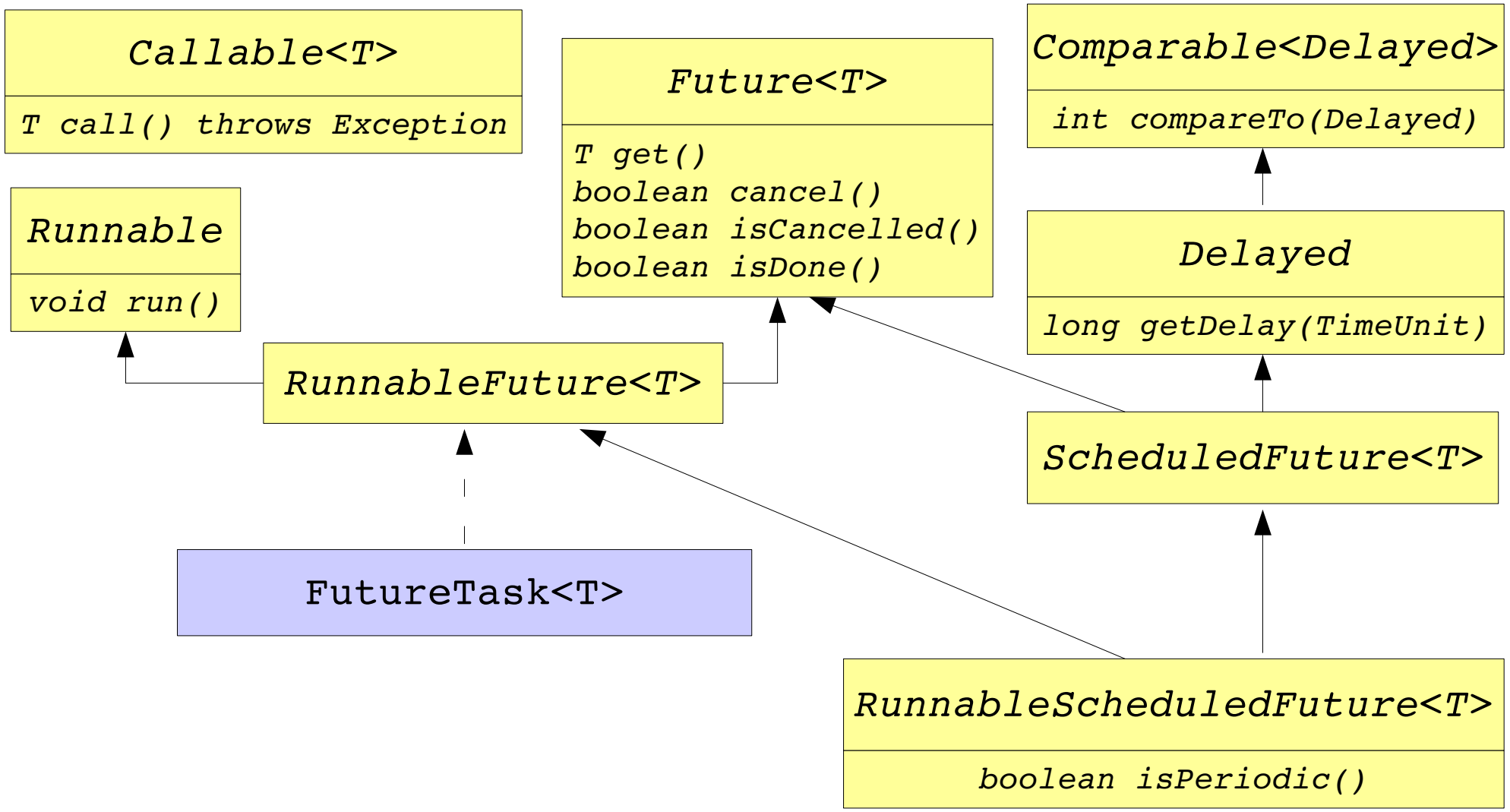
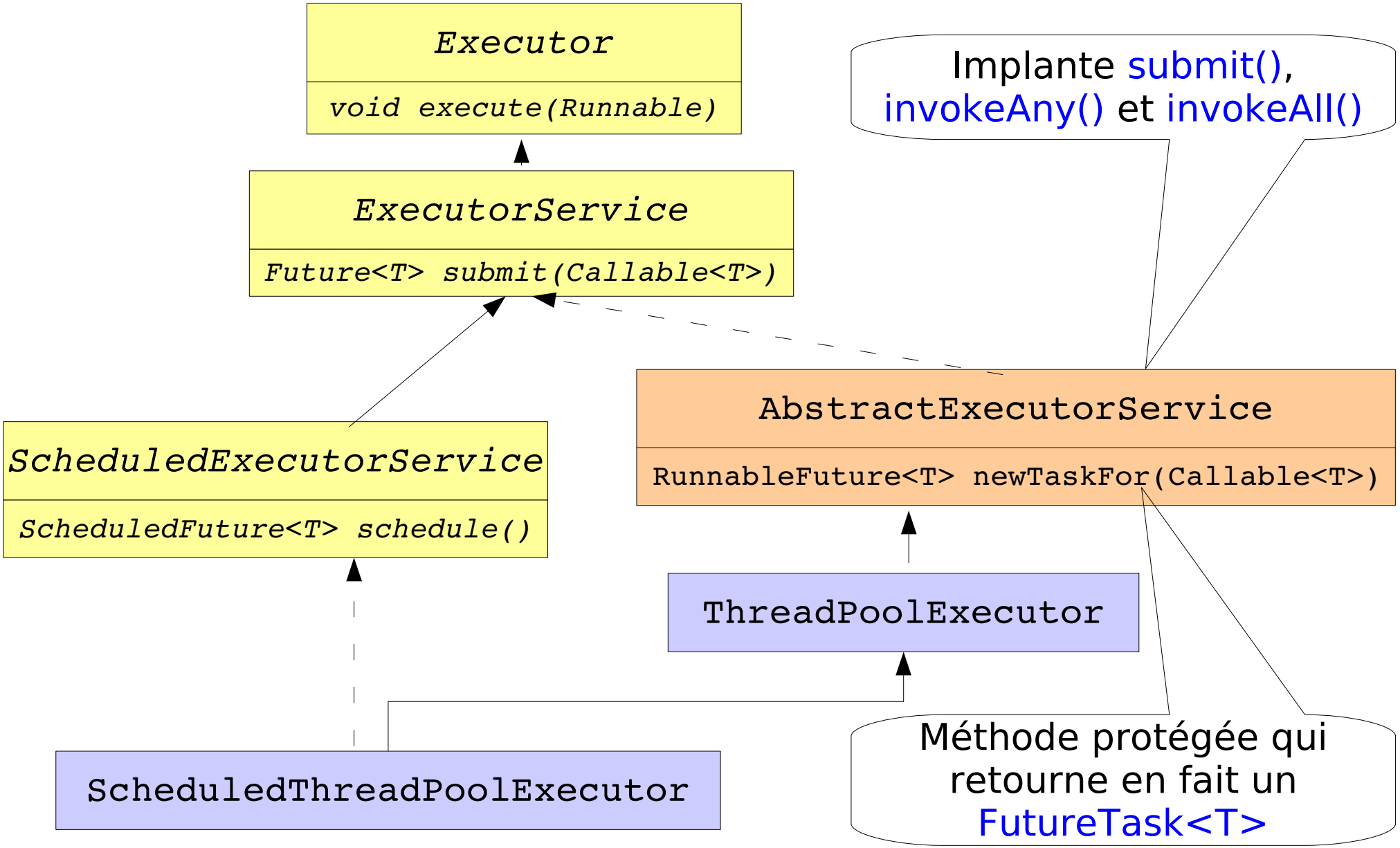
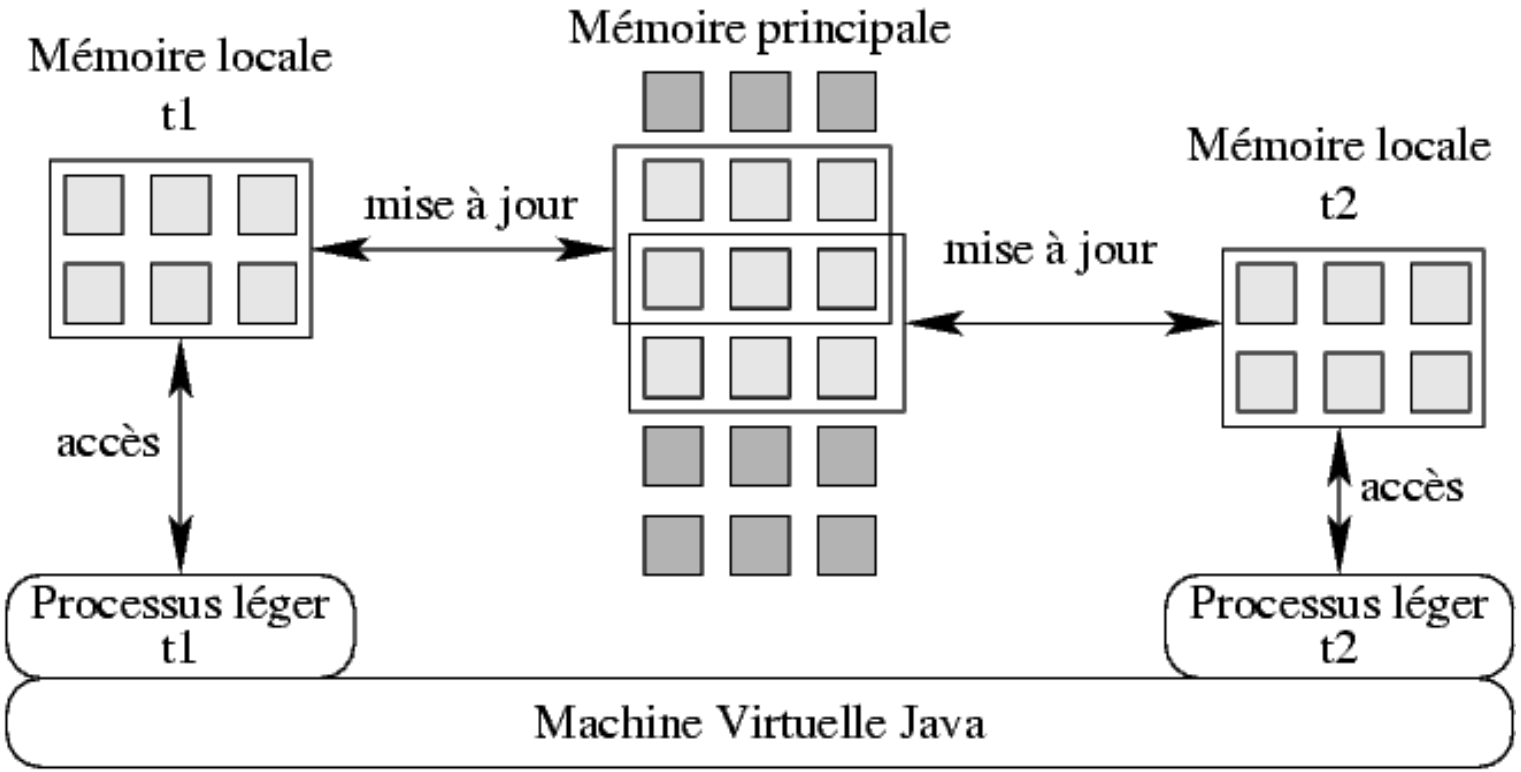
- Objet retourné par méthode exec() de Runtime
  - ```
    Process fils=
    Runtime.getRuntime().exec("commande");
    fils.waitFor();  // attend la terminaison du
    processus fils
    System.out.println(fils.exitValue());
    ```
  - Toutes les méthodes sont abstraites dans Process:
    - destroy(), getInputStream(), getOutputStream(), getErrorStream()
  - Nécessité de lire et d'écrire dans les flots d'E/S


    ```

# La classe ProcessBuilder

- Gère des "attributs de processus" communs et permet de créer plusieurs processus
  - Commande (liste de chaînes de caractères)
    - Ex: [ "javac", "-Xlint:unchecked", "Toto.java" ]
  - Environnement (Map<String, String> variables/valeurs)
    - Récupère par défaut les variables d'environnement du système (System.getenv())
    - Répertoire de travail sous la forme d'un objet File
    - Propriété de redirection de flot d'erreur (vers le flot de sortie)
      - redirectErrorStream(boolean redirect) (faux par défaut)

# La classe ProcessBuilder

- Permet de créer plusieurs Process successivement, à partir du même ProcessBuilder
  - La modification des attributs du ProcessBuilder n'affecte que les Process créés ultérieurement (pas ceux déjà créés)

```
ProcessBuilder pb =
  new ProcessBuilder("script", "arg1", "arg2");
Map<String, String> env = pb.environment();
env.put("newVariable", "newValue");
List<File> dirList = ...;
List<Process> procList = ...;
for(File workDir : list) {
    pb.directory(workDir);
    procList.add(pb.start());
}// démarre un nouveau Process pour chaque workDir
```

# Les processus légers (threads)

- Étant donnée une exécution de Java (une JVM)
  - **un seul processus** (au sens système d'expl)
  - disposer de **multiples** fils d'exécution (**threads**) internes
  - possibilités de **contrôle** plus fin (priorité, interruption...)
  - c'est la JVM qui assure l'**ordonnancement** (concurrence)
  - espace **mémoire commun** entre les différents threads
  - Deux instructions d'un même processus léger doivent en général respecter leur séquencement (sémantique)
  - Deux instructions de deux processus légers distincts n'ont pas *a priori* d'ordre d'exécution à respecter (entre-eux)

# Threads de base (exécution de `%java Prog`)

- La JVM démarre plusieurs threads, dont, par exemple:
  - "Signal Dispatcher", "Finalizer", "Reference Handler", "main"...
  - on peut avoir ces informations en envoyant un signal QUIT au programme (**Ctrl-** sous Unix ou **Ctrl-Pause** sous Windows).
  - La commande jconsole permet de "monitorer" les programmes Java lancés par java -Dcom.sun.management.jmxremote ...
- La thread "main" est chargée d'exécuter le code de la méthode main()
- Ce code peut demander la création d'autres threads
- Les autres threads servent à la gestion de la JVM (gc...)
- Exemple simple: boucle infinie dans le main() .../...

```
Full thread dump Java HotSpot(TM) Client VM (1.5.0_05-b05 mixed mode, sharing):
"Low Memory Detector" daemon prio=1 tid=0x080a4208 nid=0x2899
    runnable [0x00000000..0x00000000]
"CompilerThread0" daemon prio=1 tid=0x080a2d88 nid=0x2898
    waiting on condition [0x00000000..0x45c9df24]
"Signal Dispatcher" daemon prio=1 tid=0x080a1d98 nid=0x2897
    waiting on condition [0x00000000..0x00000000]
"Finalizer" daemon prio=1 tid=0x0809b2e0 nid=0x2896
    in Object.wait() [0x45b6a000..0x45b6a83c]
      at java.lang.Object.wait(Native Method)
      - waiting on <0x65950838> (a java.lang.ref.ReferenceQueue$Lock)
      at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:116)
      - locked <0x65950838> (a java.lang.ref.ReferenceQueue$Lock)
      at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:132)
      at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:159)
"Reference Handler" daemon prio=1 tid=0x0809a600 nid=0x2895
    in Object.wait() [0x45aea000..0x45aea6bc]
      at java.lang.Object.wait(Native Method)
      - waiting on <0x65950748> (a java.lang.ref.Reference$Lock)
      at java.lang.Object.wait(Object.java:474)
      at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:116)
      - locked <0x65950748> (a java.lang.ref.Reference$Lock)
"main" prio=1 tid=0x0805ac18 nid=0x2892
    runnable [0xbfffd000..0xbfffd4e8]
      at fr.umlv.td.test.Toto.main(Toto.java:5)
"VM Thread" prio=1 tid=0x08095b98 nid=0x2894
    runnable
"VM Periodic Task Thread" prio=1 tid=0x080a5740 nid=0x289a
    waiting on condition
```

# La classe java.lang.Thread

- Chaque instance de la classe Thread possède:
  - un nom, [get/set]Name(), un identifiant
  - une priorité, [get/set]Priority(),
    - les threads de priorité haute sont exécutées + souvent
    - trois constantes prédéfinies: [MIN / NORM / MAX]_PRIORITY
  - un statut *daemon* (booléen), [is/set]Daemon()
  - un groupe, de classe ThreadGroup, getThreadGroup()
    - par défaut, même groupe que la thread qui l'a créée
  - une cible, représentant le code que doit exécuter ce processus léger. Ce code est décrit par la méthode

    public void run() {...}

    qui par défaut ne fait rien (return;) dans la classe Thread

# Threads et JVM

- La Machine Virtuelle Java continue à exécuter des threads jusqu'à ce que:
  - soit la méthode exit() de la classe Runtime soit appelée
  - soit toutes les threads non marquées "*daemon*" soient terminées.
    - on peut savoir si une thread est terminée via la méthode isAlive()
- Avant d'être exécutées, les threads doivent être créés: Thread t = new Thread(...);
- Au démarrage de la thread, par t.start();
  - la JVM réserve et affecte l'espace mémoire nécessaire avant d'appeler la méthode run() de la cible.

# La thread courrante

```
public class ThreadExample {
  public static void main(String[] args)
                        throws InterruptedException {
    Thread t = Thread.currentThread();
    // Affiche caractéristiques de la thread courante
    System.out.println(t);
    // Lui donne un nouveau nom
    t.setName("Médor");
    System.out.println(t);
    // Rend le processus léger courant
    // inactif pendant 1 seconde
    Thread.sleep(1000);
    System.out.println("fin");
  }
}
```

```
% java ThreadExample
Thread[main,5,main]
Thread[Médor,5,main]
fin
%
```



nom priorité groupe

# Deux façons pour spécifier run()

- Redéfinir la méthode run() de la classe Thread
  - ```
    class MyThread extends Thread {
      public @Override
      void run() { /* code à exécuter*/ }
    }
    ```
  - Création et démarrage de la thread comme ceci:

    ```
    MyThread t = new MyThread();
    ```
    puis
    ```
    t.start();
    ```
- Implanter l'interface Runnable
  - ```
    class MyRunnable implements Runnable {
      public void run() { /* code à exécuter */ }
    }
    ```
  - Création et démarrage de la thread via un **objet cible**:

    ```
    MyRunnable cible = new MyRunnable(); // objet cible
    ```
    ```
    Thread t = new Thread(cible);
    ```
    puis
    ```
    t.start();
    ```

# Comparaison des deux approches

- Pas d'héritage multiple de classes en Java: hériter d'une autre classe?
- Pouvoir faire exécuter un **même** Runnable à plusieurs threads



# Par héritage de Thread

```
public class MyThread extends Thread {
  public void run() {
    for (int i=0; i<5; i++) {
      System.out.println("MyThread, en " + i);
      try {Thread.sleep(500);}
      catch (InterruptedException ie) {ie.printStackTrace();}
    }
    System.out.println("MyThread se termine");
  }
}
public class Prog1 {
  public static void main(String[] args)
                    throws InterruptedException {
    Thread t = new MyThread(); t.start();
    for (int i=0; i<5; i++) {
      System.out.println("Initial, en " + i);
      Thread.sleep(300);
    }
    System.out.println("Initial se termine");
}}
```

```
% java Prog1
Initial, en 0
MyThread, en 0
Initial, en 1
MyThread, en 1
Initial, en 2
Initial, en 3
MyThread, en 2
Initial, en 4
MyThread, en 3
Initial se termine
MyThread, en 4
MyThread se termine
```

# Par implantation de Runnable

```
public class MyRunnable implements Runnable {
  public void run () {
    for (int i=0; i<5; i++) {
      System.out.println("MyRunnable, en " + i);
      try { Thread.sleep(500); }
      catch (InterruptedException ie){ie.printStackTrace();}
    }
    System.out.println("MyRunnable se termine");
}}
public class Prog2 {
  public static void main(String[] args)
                  throws InterruptedException {
    MyRunnable cible = new MyRunnable();
    Thread t = new Thread(cible);  t.start();
    for (int i=0; i<5; i++) {
      System.out.println("Initial, en " + i);
      Thread.sleep(300);
    }
    System.out.println("Initial se termine");
}}
```

```
% java Prog2
 Initial, en 0
 MyRunnable, en 0
 Initial, en 1
 MyRunnable, en 1
 Initial, en 2
 Initial, en 3
 MyRunnable, en 2
 Initial, en 4
 MyRunnable, en 3
 Initial se termine
 MyRunnable, en 4
 MyRunnable se termine
```

# Thread et objet de contrôle

- À la fin de l'exécution de la méthode run() de la cible, le processus léger est terminé (mort):
  - il n'est plus présent dans la JVM (en tant que thread)
  - mais l'objet contrôleur (de classe Thread) existe encore
  - sa méthode isAlive() retourne false
  - il n'est pas possible d'en reprendre l'exécution
  - l'objet contrôleur sera récupéré par le ramasse-miettes
- L'objet représentant la thread qui est actuellement en train d'exécuter du code peut être obtenu par la méthode statique Thread.currentThread()

# Cycle de vie d'un processus léger

- Création de l'objet contrôleur: t = new Thread(...)
- Allocation des ressources: t.start()
- Début d'exécution de run()
  - [éventuelles] suspensions temp. d'exéc: Thread.sleep()
  - [éventuels] relâchements voulus du proc. : Thread.yield()
  - peut disposer du processeur et s'exécuter
  - peut attendre le proc. ou une ressource pour s'exécuter
- Fin d'exécution de run()
- Ramasse-miettes sur l'objet de contrôle

# L'accès au processeur

- Différents états possibles d'une thread
  - exécute son code cible (elle a accès au processeur)
  - attend l'accès au processeur (mais pourrait exécuter)
  - attend un événement particulier (pour pouvoir exécuter)
- L'exécution de la cible peut libérer le processeur
  - si elle exécute un yield() (demande explicite)
  - si elle exécute une méthode bloquante (sleep(), wait()...)
- Sinon, c'est l'ordonnanceur de la JVM qui répartit l'accès des threads au processeur.
  - utilisation des éventuelles priorités

# Différents états d'un processus léger

- Depuis 1.5, il est possible de connaître l'état d'un processus léger *via* la méthode getState(), exprimé par un type énuméré de type Thread.State :
  - NEW : pas encore démarré;
  - RUNNABLE : s'exécute ou attend une ressource système, par exemple le processeur;
  - BLOCKED : est bloqué en attente d'un moniteur;
  - WAITING : attente indéfinie de qq chose d'un autre PL;
  - TIMED_WAITING : attente bornée de qq chose d'un autre PL ou qu'une durée s'écoule;
  - TERMINATED : a fini d'exécuter son code.

# Arrêt d'un processus léger

- Les méthodes stop(), suspend(), resume() sont dépréciées
  - Risquent de laisser le programme dans un "sale" état !
- La méthode destroy() n'est pas implantée
  - Spécification trop brutale: l'oublier
- Seule manière: terminer de manière **douce**...
- Une thread se termine normalement lorsqu'elle a terminé d'exécuter sa méthode run()
  - obliger **proprement** à terminer cette méthode

# Interrompre une thread

- La méthode interrupt() appelée sur une thread t
  - Positionne un « statut d'interruption »
  - Si t est en attente parce qu'elle exécute un wait(), un join() ou un sleep(), alors ce statut est réinitialisé et la thread reçoit une InterruptedException
  - Si t est en attente I/O sur un canal interruptible (java.nio.channels.InterruptibleChannel), alors ce canal est fermé, le statut reste positionné et la thread reçoit une ClosedByInterruptException *(on y reviendra)*
- Le statut d'interruption ne peut être consulté que de deux manières, par des méthodes (pas de champ)

.../...

# Consulter le statut d'interruption

- public **static** boolean interrupted()
  - retourne true si le statut de la thread **actuellement exécutée** a été positionné (méthode statique)
  - si tel est le cas, **réinitialise** ce statut à false
- public boolean isInterrupted()
  - retourne true si le statut de la thread **sur laquelle est appelée la méthode** a été positionné (méthode d'instance, non statique)
  - ne **modifie pas** la valeur du statut d'interruption

# Exemple d'interruption

```
import java.io.*;
public class InterruptionExample implements Runnable {
  private int id;
  public InterruptionExample(int id) {
    this.id = id;
  }
  public void run() {
    int i = 0;
    while (!Thread.interrupted()) {
      System.out.println(i + "i° exécution de " + id);
      i++;
    }
    System.out.println("Fin d'exéc. du code " + id);
    // L'appel à interrupted() a réinitialisé
    // le statut d'interruption
    System.out.println(Thread.currentThread()
            .isInterrupted()); // Affiche: false
}}
```

# Exemple d'interruption

```
public static void main(String[] args) throws IOException {
  Thread t1 = new Thread(new InterruptionExample(1));
  Thread t2 = new Thread(new InterruptionExample(2));
  t1.start();
  t2.start();
  // Lecture du numéro du processus léger à interrompre
  Scanner sc = new Scanner(System.in);
  switch (sc.nextInt()) {
  case 1:
    t1.interrupt(); break;
  case 2:
    t2.interrupt(); break;
  }
}
```

```
% java InterruptionExample
 1ième exécution de 1
 2ième exécution de 1
 ...
 1ème exécution de 2
 2ième exécution de 2
 ...
          Tape 2
 ...
 Fin d'exécution du code 2
 false
 7ième exécution de 1
 8ième exécution de 1
 9ième exécution de 1
 ...
```

# Exemple d'interruption

- Objectif: ralentir l'affichage pour que chaque thread attende un temps aléatoire
  - Proposition: écrire une méthode tempo(), et l'appeler dans la boucle de la méthode run() :

```
while (!Thread.interrupted()) {
        System.out.println(i + "ième exécution de " + id);
        i++;
        tempo(); // pour ralentir les affichages
}
```

# Exemple d'interruption

- La méthode tempo() :

```
public void tempo() {
  java.util.Random rand = new Random();
  try {
    Thread.sleep(rand.nextInt(10000));
  } catch (InterruptedException ie) {
    // La levée de l'exception a
    // réinitialisé le statut
    // d'interruption. Il faut donc
    // réinterrompre le processus
    // léger courant pour repositionner
    // le statut d'interruption.
    System.out.println(Thread.currentThread().isInterrupted());
    // Affiche: false
    Thread.currentThread().interrupt();
    System.out.println(Thread.currentThread().isInterrupted());
    // Affiche: true
  }
}
```

```
% java InterruptionExample
 0ième exécution de 1
 Taper le n° de la thread
 0ième exécution de 2
 1ième exécution de 2
 1ième exécution de 1
 2ième exécution de 2
 2ième exécution de 1
 3ième exécution de 2
 3ième exécution de 1
          Tape 1
 4ième exécution de 1
 false
 true
 Fin d'exécution du code 1
 false
 4ième exécution de 2
 5ième exécution de 2
 6ième exécution de 2
 7ième exécution de 2
```

# Java.util.concurrent (depuis jdk1.5)

- Interface Executor: permet de "gérer" des threads
  - Dissocier la soumission de l'exécution des tâches
  - Une seule méthode: void execute(Runnable command)
  - Plein d'implantations: Thread-pool, IO asynchrones...
  - Différentes fabriques d'Executor dans la classe Executor**s**
  - Selon la classe concrète utilisée, la tâche planifiée est prise en charge
    - par un **nouveau** processus léger
    - ou par un processus léger "**réutilisé**"
    - ou encore par **celui qui appelle** la méthode execute()
    - et ce en **concurrence** ou de manière **séquentielle**...

# Principe d'un Executor

- Une tâche (du code à exécuter) est "soumise"
- L'Executor la "gère" (exécution, mise en file d'attente, planification, création de threads...)
- Elle est prise en charge par un "worker thread"



# Interface ExecutorService extends Executor

- Gérer plus finement la planification des commandes
- Idée: étendre les fonctionnalités offertes
  - par l'objet représentant la cible du code à exécuter: Runnable devient une nouvelle interface Callable<T>
  - par la méthode qui spécifie le code:
    - possibilité de type de retour différent de void
    - possibilité de déclarer des Exceptions propagées
  - par l'exécuteur qui permet:
    - de gérer la planification des différentes tâches, leur terminaison, leur valeur de retour, la terminaison de l'exécuteur, etc.

# Interface Callable<T>

- Callable<T> étend la notion de Runnable, en spécifiant un type de retour (de type T) et des exceptions:

```
package java.util.concurrent;
public interface Callable<T> {
  public T call() throws Exception;
}
```

- La classe d'utilitaires Executors contient des méthodes statiques permettant d'obtenir une forme "*Callable*" à partir d'autres objets représentant des tâches, par ex:
  - `Callable<Object> callable(Runnable task)`
    - Le *callable* retourne null
  - `<T> Callable<T> callable(Runnable task,T result)`
    - Le *callable* retourne result

# Soumission dans un ExecutorService

- ExecutorService ajoute trois méthodes submit()

- `<T> Future<T> submit(Callable<T> task)`

- `Future<?> submit(Runnable task)`
  - Pour ce cas, le future contiendra null
- `<T> Future<T> submit(Runnable task, T result)`
  - On peut passer un objet résultat (de type T) à retourner

  - Toutes lèvent RejectedExecutionException si la tâche ne peut pas être planifiée par l'éxécuteur, et NullPointerException si la tâche vaut null
  - Future<T> représente le résultat à venir d'un calcul asynchrone

# Interface Future<T>

- `boolean cancel(boolean mayInterruptIfRunning)`
  - Retourne false si pas annulable (déjà terminée)
- `boolean isCancelled()`: annulée avant d'avoir terminée
- `boolean isDone()`: terminée (quelle que soit la manière)
- `T get() throws InterruptedException, ExecutionException, CancellationException`
  - Méthode bloquante jusqu'à:
    a) terminaison normale; b) thread interrompue;
    c) levée d'exception à l'exécution; d) tâche annulée;
- `T get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException, CancellationException`
  - Idem, mais si timeout est écoulé, lève TimeoutException

# Classes et interfaces autour de Future<T>



# Les méthodes de ExecutorService

- void **shutdown**() continue à exécuter les tâches déjà planifiées, mais plus aucune tâche ne peut être soumise
- List<Runnable> **shutdownNow**() tente d'arrêter activement (typiquement Thread.interrupt()) et/ou n'attend plus les tâches en cours, et retourne la liste de celles qui n'ont pas commencé
- boolean **isShutdown**() vrai si **shutdown**() a été appelé
- boolean **isTerminated**() vrai si toutes les tâches sont terminées après un shutdown() ou shutdownNow()
- boolean **awaitTermination**(long timeout, TimeUnit unit) bloque jusqu'à ce que:
  - soit toutes les tâches soient terminées (après un shutdown(),
  - soit le timeout ait expiré,
  - soit la thread courante soit interrompue

# Exemple ExecutorService

```
public static void main(String[] args) throws Throwable {
  final InetAddress host=
                InetAddress.getByName("gaspard.univ-mlv.fr");
  ExecutorService executor = Executors.newCachedThreadPool();
  Future<Long> future = executor.submit( // création et
    new Callable<Long>() {                // soumission d'une
      public Long call() throws Exception {  // tâche qui
        long t = System.nanoTime();       // retourne un long
        if(!host.isReachable(2000))
          throw new UnreachableException(host.toString());
        return System.nanoTime()-t;
      }
    });
  executor.shutdown();      // fermeture de l'Executor
  try {
    System.out.println("reach "+host+" in "+
      future.get()+" ns"); // l'appel à get est bloquant
  } catch (ExecutionException e) { throw e.getCause(); }
} // Si l'exécution lève une exception, elle est transmise
// reach gaspard.univ-mlv.fr/193.55.63.81 in 1059000 ns
```

# Les méthodes de ExecutorService

```
// Attend la fin de toutes les exécutions (succès ou exception)
// L'ordre de la liste des futures correspond à celui de tasks
<T> List<Future<T>>
    invokeAll(Collection<? extends Callable<T>> tasks)
          throws InterruptedException
<T> List<Future<T>>
    invokeAll(Collection<? extends Callable<T>> tasks,
                              long timeout, TimeUnit unit)
          throws InterruptedException

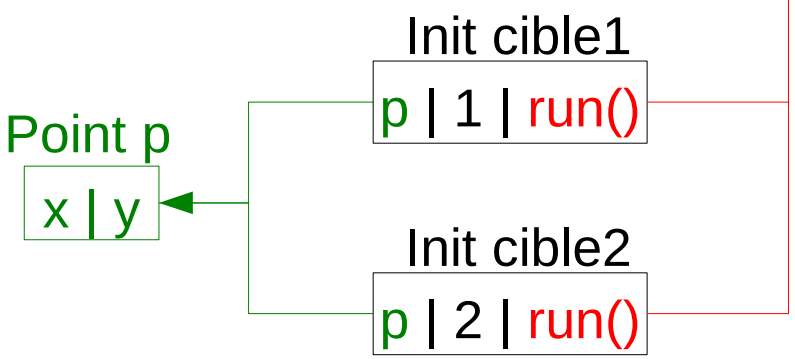
// Attend une exécution avec succès (sans levée d'exception)
// Les tâches restantes sont annulées (cancel())
<T> T invokeAny(Collection<? extends Callable<T>> tasks)
            throws InterruptedException, ExecutionException

<T> T invokeAny(Collection<? extends Callable<T>> tasks,
                  long timeout, TimeUnit unit)
            throws InterruptedException, ExecutionException,
                  TimeoutException
```

# Exemple invokeAny()

```
InetAddress[] hosts = // { zarb, strange, gaspard }
ArrayList<Callable<Long>> callables=
  new ArrayList<Callable<Long>>();
for(final InetAddress host: hosts) {
  callables.add(new Callable<Long>() {
    public Long call() throws Exception {
      long time=System.nanoTime();
      if(!host.isReachable(2000))
        throw new UnreachableException(host.toString());
      return System.nanoTime()-time;
    }
  });
}
ExecutorService e = Executors.newFixedThreadPool(2);
System.out.println("closest host is at "
  +e.invokeAny(callables)+" ns");
e.shutdown();
// Affiche: closest host is at 813000 ns
// Les exécutions qui lèvent des exceptions ne sont
// pas prises en compte et les autres tâches sont annulées
```

# Exemple invokeAll()

```
InetAddress[] hosts = // { zarb, strange, gaspard }
ArrayList<Callable<Long>> callables = ...
// avec mêmes callables et executor que précédemment
List<Future<Long>> futures = e.invokeAll(callables);
e.shutdown();
Iterator<Future<Long>> itf = futures.iterator();
for(InetAddress host : hosts) {
      System.out.print(host);
      Future<Long> f = itf.next();
      try {
        System.out.println(" is at " + f.get() + " ns");
      }
      catch(ExecutionException ee) {
        Throwable t = ee.getCause();
        if (t instanceof UnreachableException)
          System.out.println(" is unreachable");
        else throw t;
      } // Affiche: zarb is unreachable
    }   //          strange is unreachable
}       //          gaspard is at 3195000 ns
```

# Interface ScheduledExecutorService

- Ajoute des notions temporelles (planification)

```
<V> ScheduledFuture<V> schedule(Callable<V> callable,
                                long delay, TimeUnit unit)

ScheduledFuture<?> schedule(Runnable command,
                            long delay, TimeUnit unit)

ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                       long initialDelay,
                                       long period, TimeUnit unit)
                                       // tente de respecter la cadence
                                       // des débuts d'exécutions
```




```
ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
                                          long initialDelay,
                                          long delay, TimeUnit unit)
                                          // tente de respecter l'espacement
                                          // entre la fin d'une exécution
                                          // et le début de la suivante
```



# ScheduledFuture et TimeUnit

- L'interface ScheduledFuture hérite de Future mais aussi de Delayed
  - long getDelay(TimeUnit unit) qui représente le temps restant avant l'exécution (0 ou négatif signifie délai dépassé)
- TimeUnit énuméré représentant 1 durée dans une unité donnée
  - MICROSECONDS, MILLISECONDS, NANOSECONDS, SECONDS, MINUTES, HOURS, DAYS
  - Offre d'autres méthodes de conversion ou d'attente
    `public long convert(long sourceDuration,TimeUnit sourceUnit)`
    par ex: `TimeUnit.MILLISECONDS.convert(10L, TimeUnit.MINUTES)`
    convertit 10 minutes en milisecondes
  - Raccourcis: `public long toNanos(long duration)`
    équivalent à: `NANOSECONDS.convert(duration, this)`

# Autour de Future<T> et des planifications



# Autour de Executor et ExecutorService



# Les implantations accessibles d'Executors

- ThreadPoolExecutor
  - Normalement configuré par les fabriques de Executors
    - Executors.newCachedThreadPool() pool de thread non borné, avec réclamation automatique des threads inactives
    - Executors.newFixedThreadPool(int) pool de threads de taille fixe, avec file d'attente non bornée
    - Executors.newSingleThreadExecutor() une seule thread qui exécute séquentiellement les tâches soumises
  - Si une thread d'un exécuteur meurt, elle est remplacée
  - ScheduledThreadPoolExecutor
    - À utiliser de préférence plutôt que java.util.Timer
    - Pool de **taille fixe** et file d'attente non bornée
- Méthodes statiques pour encapsuler un Executor ou un ScheduledExecutor (cache l'implémentation): `ExecutorService unconfigurableExecutorService(ExecutorService)`

# Interface CompletionService et implantation

- Permet de découpler la planification d'un ensemble de tâches de la manière dont on gère leurs résultats
  - Interface CompletionService<T>
  - implementation ExecutorCompletionService<T>
    - Constructeur accepte un Executor en argument
  - Pour soumettre une tâche:
    `Future<T> submit(Callable<T> task)` ou
    `Future<T> submit(Runnable task, T result)`
  - Pour récupérer (bloquant) et "retirer" les résultats (dans l'ordre): `Future<V> take() throws InterruptedException`
  - Récupérer/retirer (non bloquant; peut renvoyer null)

    `Future<V> poll()` ou
    `Future<V> poll(long timeout, TimeUnit unit) throws InterruptedException`

# Exemple CompletionService

```
// InetAddress[] hosts = // { zarb, strange, gaspard }
// Mêmes callables que dans les autres exemples
InetAddress[] hosts = // { zarb, strange, gaspard }
ArrayList<Callable<Long>> callables = ...
// avec mêmes callables et executor que précédemment
ExecutorService e = Executors.newFixedThreadPool(3);
CompletionService<Long> cs =
  new ExecutorCompletionService<Long>(e);
for(Callable<Long> task : callables) {
  cs.submit(task);
}
e.shutdown();
System.out.println("closest host is at " + cs.take().get());
// Attention: contrairement à invokeAny()
//   - le get() peut lever une ExecutionException
//   - les autres tâches continuent à s'exécuter
```

# Principe de la programmation concurrente

- Les différents threads ont des piles d'exécution et des registres propres, mais accèdent à un même tas (espace mémoire de la JVM)
- Trois types d'objets/variables:
  - Ceux qui sont consultés/modifiés par 1 seul thread
    - Pas de problème, même s'ils résident dans la mémoire centrale
  - Ceux qui sont gérés **localement** au niveau d'un thread
    - Classe dédiée à ce type: **ThreadLocal**
    - Typiquement: la valeur d'un champ (de ce type) d'un objet partagé entre plusieurs threads est locale à chaque thread: pas de problème d'accès concurrent
  - Ceux qui sont **accédés en concurrence**

# Variables locales à une thread

- Si plusieurs threads exécutent le même objet cible (de type Runnable), on peut vouloir disposer de variables locales propres à chaque thread.
- ThreadLocal<T> et InheritableThreadLocal<T> permettent de simuler ce comportement
  - objet déclaré comme un champ dans l'objet Runnable
  - existence d'une valeur encapsulée (sous-type de Object) propre à chaque thread, accessible via les méthodes get() et set()

# Exemple

```
import java.util.Random;
public class CodeWithLocal implements Runnable {
  public ThreadLocal<Integer> local = new ThreadLocal<Integer>();
  public Object shared;
  public void run() {
    String name = Thread.currentThread().getName();
    Random rand = new Random();
    local.set(rand.nextInt(100));
    System.out.println(name + ": locale: " + local.get());
    shared = rand.nextInt(100);
    System.out.println(name + ": partagée: " + shared);
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    }
    System.out.println(name + ": après attente, locale: " +
                                                  local.get());
    System.out.println(name + ": après attente, partagée: " +
                                                  shared);
  }
}
```

# Exemple (suite)

```
public class ExecCodeWithLocal {
  public static void main(String[] args) throws InterruptedException {
    // Création d'un objet code exécutable (cible)
    CodeWithLocal code = new CodeWithLocal();
    // Création de deux processus légers ayant ce même objet pour cible
    Thread t1 = new Thread(code);
    Thread t2 = new Thread(code);
    // Démarrage des processus légers
    t1.start();
    t2.start();
    // Affichage des champs de l'objet
    // cible, après la fin des exécutions,
    // depuis le processus léger initial
    t1.join();
    t2.join();
    System.out.println("Initial: locale: "
                       + code.local.get());
    System.out.println("Initial: partagée:"
                       + code.shared);
  }
}
```

```
% java ExecCodeWithLocal
Thread-0: locale: 92
Thread-0: partagée: 43
Thread-1: locale: 11
Thread-1: partagée: 41
Thread-0: après attente, locale: 92
Thread-0: après attente, partagée: 41
Thread-1: après attente, locale: 11
Thread-1: après attente, partagée: 41
Initial: locale: null
Initial: partagée:41
%
```

# Transmission des variables locales

- La classe InheritableThreadLocal<T> permet de transmettre une variable locale à une thread t1 à une thread t2 « fille », c'est-à-dire créée à partir de t1 (de la même façon qu'elle récupère sa priorité, son groupe, etc.).
  - initialisation des champs de classe InheritableThreadLocal<T> de la thread « fille » à partir des valeurs des champs de la thread «mère», par un appel implicite à la méthode T childValue(T) de la classe.
  - possibilité de redéfinir cette méthode dans une sous-classe de InheritableThreadLocal<T>.

# Problèmes de la concurrence

- Le programmeur n'a pas la main sur l'architecture des processeurs, ni sur le scheduler de l'OS.
  - Problèmes pour un thread qui regarde des variables modifiées par un autre thread (***visibilité***)
  - Le scheduler est preemptif
    - Il arrête les thread où il veut (opération atomique)
- Les instructions des différents threads peuvent, *a priori*, être exécutées dans n'importe quel ordre.
  - Notion d'**indépendance** des traitements?
  - Organiser les threads entre-elles, "synchronisation", rendez-vous, exclusion mutuelle, échange...
- Nécessité d'assurer la **cohérence** des données
  - En imposant un contrôle sur les lectures/éecriture
  - Peut provoquer des famines ou des inter-blocages

# Modèle d'exécution

- La garantie du modèle d'exécution :
  - Deux instructions d'un même processus léger doivent respecter leur séquencement si elles dépendent l'une de l'autre

```
a=3;
c=2;
b=a; // a=3 doit être exécuté avant b=a
```

  - Mais on ne sait pas si **c=2** s'exécute avant ou après **b=a** (réordonnancements, optimisations)
- De plus, deux instructions de deux processus légers distincts n'ont pas *a priori* d'ordre d'exécution à respecter.

# Problèmes liés au processeur

- Le JIT
  - stocke les variables dans des registres pour éviter les aller-retour à la mémoire (tas)
  - ré-ordonnance les instructions à l'intérieur d'un thread pour de meilleures performances (pipeline)
- Les long et double sont chargés en 2 fois sur une machine 32bit
  - Pas de problème pour le thread courant
  - Problème pour un autre thread qui regarde (accède aux mêmes variables): **cohérence**

# Visibilité et mise à jour des variables

- Chaque processus léger stocke les valeurs qu'il utilise dans ses registres: la mise à jour n'est pas systématique



# Les registres

- Deux threads:
  - Le premier exécute tant qu'une valeur est false

```
public class A implements Runnable {
  public boolean start;  // false
  public void run() {
    while(!this.start) ; // attente active
    ...
  }
}
```

Ici, la valeur false est mise dans un registre

  - Au bout d'un temps, l'autre thread change la valeur

```
public void doSomething(A a) {
  // ...
  a.start=true;
}
```

Ici, la valeur true est mise en mémoire, mais la première thread ne **recharge pas** cette valeur dans son **registre**

  - Essayer avec java -server

# Registres et réordonnacement

```
public class A implements Runnable {
  public boolean start;  // false
  public int value;
  public void run() {
    while(!this.start) ; // attente active
    System.out.println(value); // affiche 0
  }
}
```

```
public void doSomething(A a) {
  a.value = 3;
  a.start=true;
  f(value);
}
```

Réordonnancement

```
a.start=true;
a.value = 3;
f(value);
```

# Solution: le mot clé volatile

- Utilisé comme un modificateur de champ
- Assure la **cohérence** entre la mémoire de travail et la mémoire principale (pas de cache possible).
  - La mise à jour est forcée à chaque lecture/écriture pour prendre en compte les dernières modifications
- L'**écriture** d'une variable **volatile** force la **sauvegarde de tous les registres** dans la mémoire
- Impose une **barrière mémoire**
  - Interdit le réordonnancement entre les variables volatiles et les autres
- Assure l'**atomicité** de la lecture et de l'écriture des variables de type double et long

# Le mot-clef **volatile**

- Attention: n'assure **pas l'atomicité** des opérations composites
  - Y compris l'**incrémentation** ou la **décrémentation**.
  - Les instructions composant ces opérations peuvent s'entrelacer entre plusieurs threads concurrents
  - Ex:
    1) Un champ x d'un objet o, déclaré volatile, vaut 0;
    2) Deux threads concurrents font o.x++; sur o;
    3) Il peut arriver que, une fois les 2 threads exécutées, (o.x == 1) soit vrai!
    (et non 2 comme attendu)

# Publication d'un objet

- Il est possible d'accéder à la référence d'un objet avant qu'il soit fini de construire
  - Avant que son initialisation soit terminée

```
public class A implements Runnable {
  public int value;
  public A(int value) {
    this.value = value;
  }
}

final A a=new A(3);
new Thread(new Runnable() {
  public void run() {
    System.out.println(a.value);
  }
}).start();
```

```
1: A = new A     // allocation
2: a.value = 3; // initialisation
3: new Thread(...).start();
```

```
// peut afficher 0
// Si réordonnancement
// et scheduling entre 1 et 2
```

# Champ final changeant de valeur:-(

- Les valeurs d'un champ final sont assignées dans le constructeur de la classe avant la publication de la référence
  - Une fois construit, le champ final peut être accédé par différents threads sans synchronisation
- Mais pendant sa construction, réordonancement et publication explicite sont possibles...

```
public class A {
   public final int value;
   public A(B b,int value) {
     this.value=value;
     b.a=this; // beurk
}

public class B {
  public A a;
}
```

```
// et dans un autre thread
B b = ...
System.out.println(b.a.value);
                    // peut afficher 0
```

# Règles à respecter

- **Ne pas "publier"** (exposer explicitement) la référence sur **this** dans le **constructeur**
- Ne pas **exposer implicitement** la **référence this** dans le constructeur
  - Par exemple, l'exposition ou la publication de la référence à une classe interne non statique contient une référence implicite à this
- Ne pas démarrer de thread (start()) à l'intérieur d'un constructeur
  - Mieux vaut équiper la classe de sa propre méthode start() qui la démarre (après achèvement de la construction)

# Encore un problème

```
public class Point {
  private int x;
  private int y;
  public void change(int x,int y) {
    this.x = x; // Scheduling ici
    this.y = y;
  }
  public String toString() {
    return "("+x+","+y+")";
  }
}
```

```
public class StayHere implements
                          Runnable{
  private final Point p;
  private int val; // coordonées
  public StayHere(Point p,int v){
    this.p = p;
    this.val = v;
  }
  public void run() {
    for(;;) {
      p.change(val,val);
      System.out.println(p);
    }
  }
  public static void main(...) {
    Point p = new Point();
    Init cible1 = new StayHere(p,1);
    Init cible2 = new StayHere(p,2);
    Thread t1 = new Thread(cible1);
    Thread t2 = new Thread(cible2);
    t1.start();
    t2.start();
    // peut afficher (1,2)
    // ou (2,1)
  }
}
```



# Exclusion Mutuelle

- Écriture/lecture entrelacées provoquent des états incohérents de la mémoire
- Atomicité: Java garantit l'atomicité de l'accès et de l'affectation aux champs de tout type sauf double et long (64 bits)
- **Volatile** et atomics (java.util.concurrent.atomic) ne peuvent rien à l'exemple précédent
- Impossible d'assurer qu'un thread ne « perdra » pas le processeur entre deux instructions atomiques
- On ne peut « que » **exclure mutuellement** plusieurs threads, grâce à la notion de **moniteur**

# Les moniteurs

- N'importe quel objet (classe Object) peut jouer le rôle de moniteur.
  - Lorsqu'un thread « prend » le moniteur associé à un objet, plus aucun autre thread ne peut prendre ce moniteur.
  - Idée: protéger les portions de code « sensibles » de plusieurs threads par le même moniteur
    - Si la thread « perd » l'accès au processeur, elle ne perd pas le moniteur
      - une autre thread ayant besoin du même moniteur ne pourra pas exécuter le code que ce dernier protège.
  - Le mot-clef est **synchronized**

# Protection de code

- ```
  synchronized (monitor) {
  // bloc protégé par monitor
  }
  ```
- Si on protège toute une méthode avec le moniteur associé à this, syntaxe modificateur de méthode

```
public synchronized void
  change(int x, int y) {
      this.x = x;
      this.y = y;
 }
```

  - Mais: **éviter de synchroniser sur this**
    - Risque d'utilisation "externe" de this comme moniteur
- Dans notre exemple, cela suffit il?
- Taille mini des blocs de code protégés (performances)

```
public class Point {
 private int x;
 private int y;
 private final Object m
             = new Object();
 void change(int x,int y) {
   synchronized (m) {
     this.x = x;
     this.y = y;
   }
 }
 public String toString() {
   return "("+x+","+y+")";
 }
}
```
```

# Liste récursive

- Imaginons une liste récursive avec une structure constante (seules les valeurs contenues peuvent changer)
- Imaginons plusieurs threads qui parcourent, consultent et modifient les valeurs d'une même liste récurisve
- On souhaite écrire une méthode calculant la somme des éléments de la liste.

# Une implantation possible

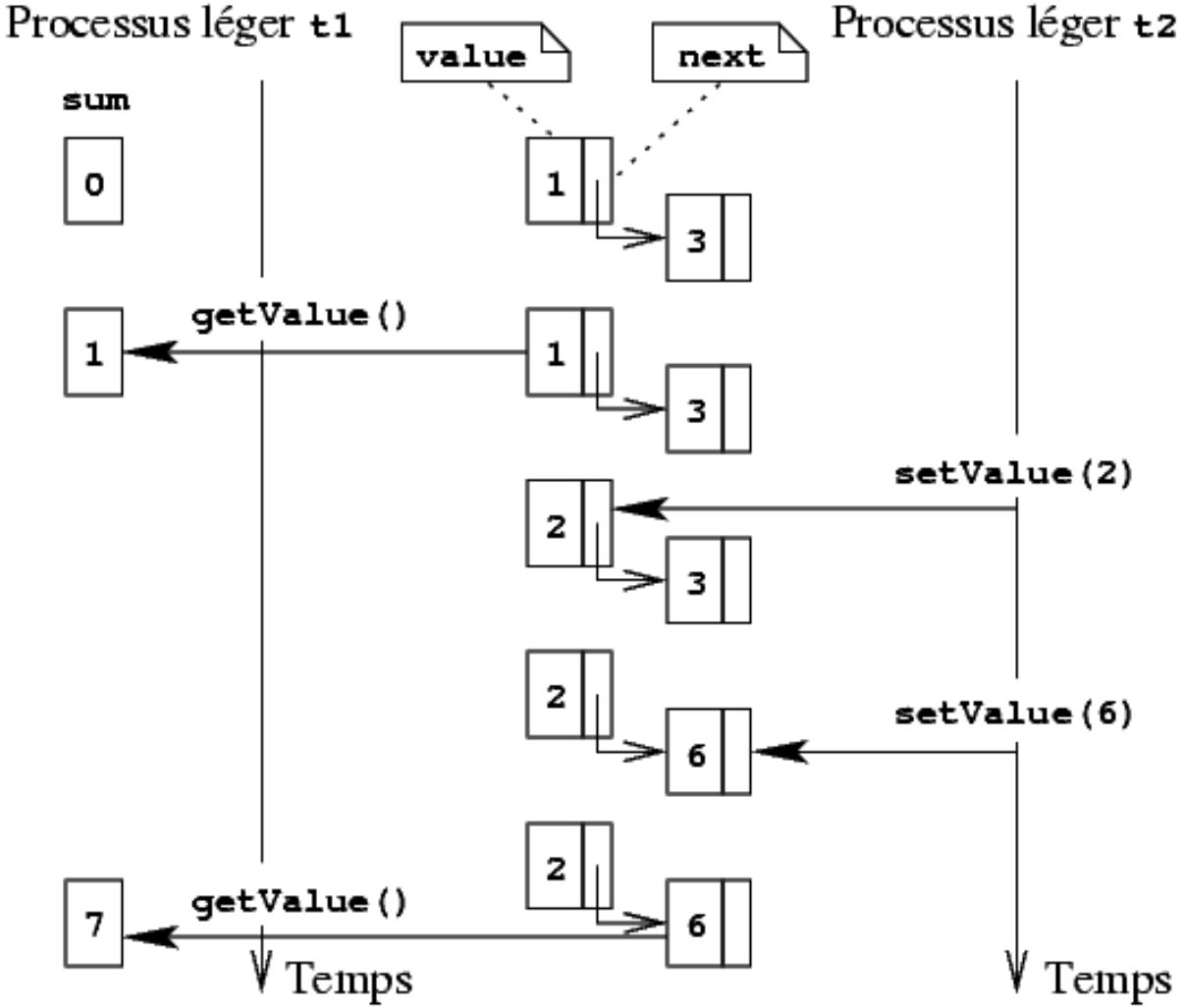
```
public class RecList {
  private double value;
  private final RecList next; // structure constante
  private final Object m = new Object(); // protection accès à value
  public RecList(double value, RecList next){
    this.value = value;
    this.next = next;
  }
  public void setValue(double value) {
    synchronized (m) { this.value = value; }
  }
  public double getValue() { synchronized (m) { return value; } }

  public RecList getNext() { return next; }

  public double sum() {
    double sum = getValue();
    RecList list = getNext();
    if (list!=null)
      sum += list.sum();
    return sum;
  }
}
```

# Mais volatile peut suffire

```
public class RecList {
  private volatile double value;
  private final RecList next; // structure constante

  public RecList(double value, RecList next){
    this.value = value;
    this.next = next;
  }
  public void setValue(double value) { this.value = value; }

  public double getValue() { return value; }

  public RecList getNext() { return next; }

  public double sum() {
    double sum = getValue();
    RecList list = getNext();
    if (list!=null)
      sum += list.sum();
    return sum;
  }
}
```

# Problème dans les deux cas

- La valeur retournée par la méthode sum() peut correspondre à une liste qui n'a jamais existé.



# Solutions?

- Protéger la méthode sum() par un **moniteur propre au maillon** sur lequel on fait l'appel
  - Le moniteur sur le premier maillon est pris et conservé jusqu'au retour de la fonction et chaque appel récursif reprend le moniteur propre au nouveau maillon.

```
public double sum() {
  synchronized (m) {
    double sum = getValue();
    RecList list = getNext();
    if (list!=null)
      sum += list.sum();
    return sum;
  }
}
```

- N'empêche pas une modification en fin de liste entre le début et la fin du calcul de la somme.
- Sum est une vue "weakly-consistant" de la liste

# Sémantique différente: plus stricte

- Pour une vue "snapshot" de la liste
  - Protéger les méthodes sur un seul et même moniteur partagé par tous les maillons d'une même liste.
  - Chaque appel récursif **reprend le moniteur** global de la liste: les moniteurs sont **réentrants**
  - Ce moniteur sert à exclure mutuellement
    - le **calcul de la somme** de la liste et
    - les **modifications des valeurs** de la liste
- Sémantique différente
  - pas forcément meilleure ou moins bonne
- Plus contraignant pour les autres threads
  - Performances *versus* sémantique

# Implantation snapshot

```
public class RecList2 {
  private volatile double value;
  private final RecList next; // structure constante
  private final Object mList; // moniteur global de la liste
  public RecList2(double value, RecList2 next) {
    this.value = value;
    this.next = next;
    if(next==null)
      mList = new Object();
    else
      mList = next.mList;
  }
  public double sum() {
    double sum;
    synchronized (mList) {
      sum = getValue();
      RecList list = getNext();
      if (list!=null)
        sum += list.sum();
    }
    return sum;
  }
```

```
  // Modification: exclusion mutuelle
  public void setValue(double value) {
    synchronized (mList) {
      this.value = value;
    }
  }
  // Consultation: excl. mut. inutile
  public double getValue() {
    return value;
  }

  public RecList2 getNext() {
    return next;
  }
}
```

# Protection en contexte statique

- Comment est protégée une méthode statique? Autrement dit, que signifie:
  - 
```
public class C {
  public static synchronized int m(...){
    ...
  }
}
```
  - Le moniteur est l'objet classe (de type java.lang.Class)
    autrement dit, c'est équivalent à:
  - 
```
public class C {
  public static int m(...) {
    synchronized (Class.forName("C")) {...}
  }
}
```

# Les moniteurs sont "réentrants"

- Si une thread **t** détient un moniteur **m**
  - Alors si **t** exécute du code qui impose une synchronisation sur le même moniteur **m**, alors c'est bon!
  - En revanche, si **t** crée et démarre une thread **tBis**, alors **tBis** ne détient pas (n'hérite pas de sa mère) le moniteur **m**, mais entre en concurrence d'accès avec la thread **t** (**tBis** devra attendre que **t** libère **m** pour espérer pouvoir le prendre)

# Inter-blocage

- Si on n'utilise pas les méthodes dépréciées (stop(), suspend(), resume()), le principal problème est l'**inter-blocage (dead lock)**.

```
public class Deadlock {
  Object m1 = new Object();
  Object m2 = new Object();
  public void ping() {
    synchronized (m1) {
      synchronized (m2) {
        // Code synchronisé sur
        // les deux moniteurs
      }
    }
  }
```

```
  public void pong() {
    synchronized (m2) {
      synchronized (m1) {
        // Code synchronisé sur
        // les deux moniteurs
      }
    }
  }
}
```

# Inter-blocage et bloc statique

- Tout bloc statique est protégé par un verrou

```
public public class DeadlockLazy {
  // les initialisations/codes statiques prennent le moniteur
  static boolean initialized = false;
  static {
    Thread t = new Thread(new Runnable() {
      public void run() {
        initialized = true;
      }
    });
    t.start(); // le thread créé ne détient pas le moniteur
    try {
      t.join();
    } catch (InterruptedException e) {
      throw new AssertionError(e);
    }
  }
  public static void main(String[] args) {
    System.out.println(initialized);
  }
}
```

# Différents aspects de la synchronisation

- **Exclusion mutelle** vis à vis d'un moniteur
  - L'objet qui sert de moniteur est «  inerte  » (la synchronisation n'a aucun effet sur lui)
- **Barrière mémoire**:
  - Le relâchement d'un moniteur (***release***) force l'écriture dans la mémoire principale de tout ce qui a été modifié dans le cache (registres) avant ou pendant le bloc synchronisé
  - L'acquisition d'un moniteur (***acquire***) invalide le cache (registres) et force la relecture depuis la mémoire principale
- **Base pour l'attente/notification** (rendez-vous)

# Atomicité *vs* exclusion mutuelle

- La notion d'**atomicité** tend à **garantir** que l'**état** de la **mémoire** sera **cohérent**
  - volatile, java.util.concurrent.atomics.*
- La notion d'**exclusion mutuelle** (synchronized) ne fait qu'assurer que du code n'aura **pas accès** à un **état** de la **mémoire incohérent**
    - La mémoire peut être dans un état incohérent et un code tierce non protégé peut y avoir accès
  - Synchronisation généralisée: Verrous et Sémaphore
    - java.util.concurrent.locks.Lock
    - java.util.concurrent.Semaphore
  - Attente notification
    - wait()/notify() sur les moniteurs
    - java.util.concurrent.locks.Condition

# Synchronisation incorrecte

- C'est lorsqu'il peut y avoir une "course aux données" (***data race***), c'est-à-dire lorsque:
  - Une thread **écrit** dans une variable
  - Une autre thread **lit** dans cette variable
  - L'**écriture** et la **lecture** ne sont **pas ordonnées** explicitement par synchronisation. Exemple:

```
public class SimpleDataRace {
  static int a = 0;
  public static void main(String[] args) {
    new Thread() {
      public void run() {
        System.out.println(a);
      }
    }.start();
    a = 1;
  } /* Peut afficher 1 comme 0. */
}
```

# Synchronisation (rendez-vous)

- « Attendre » qu'un processus léger soit dans un état particulier.
  - Attendre qu'un thread t soit terminée: t.join(); accepte éventuellement une durée (en milli/nanos)
  - Attendre l'arrivée d'un événement particulier ou notifier l'arrivée de cet événement: wait(); / notify();
- o.wait(); exécuté par un thread donnée t1
  - requiert de détenir le moniteur o
  - suspend le thread courant t1 et libère le moniteur o
  - le thread suspendu attend (passivement) d'être réveillé par une notification sur ce même moniteur
  - Lorsque t1 reçoit une notification associée à o, il doit à nouveau acquérir le moniteur associé à o afin de poursuivre son exécution.

# wait() (suite)

- wait() peut accepter un délai d'attente en argument
  - l'attente est alors bornée par ce délai,
  - passé ce délai, le thread peut reprendre son exécution mais doit auparavant reprendre le moniteur o.
- Si un autre thread interrompt l'attente: t1.interrupt();
  - une exception InterruptedException est levée par wait();
  - le statut d'interruption est réinitialisé par cette levée d'exception (il peut être nécessaire de le repositionner)

# Notification

- Une notification peut être émise par un thread t2
  - par o.notify(): un seul thread en attente de notification sur o sera réveillé (choisi arbitrairement);
  - ou par o.notifyAll(): tous les threads en attente de notification sur o seront réveillés (ils entrent alors en concurrence pour obtenir ce moniteur).
- L'exécution de o.notify() ou o.notifyAll() par t2
  - requiert que t2 **détienne le moniteur** associé à o
  - ne libère pas le moniteur (il faut attendre d'être sorti du bloc de code protégé par un synchronized(o){...}

# Attente/Notification

- Utilise la notion de moniteur
  - si wait() ou notify() sans détenir le moniteur associé à o, alors levée d'une exception IllegalThreadStateException
- Absence de tout ordonnancement connu
  - possibilité de « perdre » une notification
    - par ex. si elle a été émise par t2 avant que t1 soit mise en attente
  - possibilité d'être réveillé par une notification correspondant à un événement « consommé » par quelqu'un d'autre
  - assurer impérativement que l'événement est disponible !

# Exemple typique

- **Toujours** faire les **wait()** dans une **boucle**
  - Après le réveil et la ré-acquisition du moniteur, cela assure que la condition requise est toujours valide!
  - en particulier, elle n'a pas été consommée par un tiers

```
// code des processus légers
// intéressés par les événements
synchronized (o) {
  while (ok==0) {
    o.wait();
  }
  ok--;
}
traitement();
```

```
// code du processus léger
// qui signale les événements
synchronized (o) {
  ok++;
  o.notify();
}
```

# Les problèmes classiques

- Grâce aux mécanismes d'exclusion mutuelle et de attente/notification, on tente d'assurer que « rien de faux n'arrive ».
- Problème fréquent « rien n'arrive du tout »: c'est la notion de vivacité, qui se décline en
  - **famine**
    une thread pourrait s'exécuter mais n'en a jamais l'occasion
  - **endormissement**
    une thread est suspendue mais jamais réveillée
  - **interblocage**
    plusieurs threads s'attendent mutuellement

# Pour éviter les interblocages

- Numéroter les événements à attendre
  - toujours les attendre dans le même ordre
- Plus de concurrence améliore la vivacité mais risque d'aller à l'encontre de la « sûreté » du code dont il faut relâcher la synchronisation
  - Ce qui est acceptable dans un contexte ne l'est pas forcément toujours (réutilisabilité du code?)
- Si concurrence trop restreinte on peut parfois « diviser pour régner »
  - tableau de moniteurs plutôt que moniteur sur tableau?

# Y a t il un problème?

```
public class Stack {
  LinkedList list = new LinkedList();
   public synchronized void push(Object o) {
      synchronized (list) {
         list.addLast(o);
         notify();
      }
   }
   public synchronized Object pop()
                                 throws InterruptedException {
         synchronized (list) {
           while (list.isEmpty())
              wait();
           return list.removeLast();
         }
   }
}
```

# Oui! (en plus de la double synchronisation)

```
public class Stack {
  LinkedList list = new LinkedList();
  public synchronized void push(Object o) {
    synchronized (list) {// Ici, le moniteur est list
      list.addLast(o);
      notify();              // Mais ici, c'est notify() sur this
    }
  }
  public synchronized Object pop()
                                  throws InterruptedException {
    synchronized (list) {// Ici, le moniteur est list
      while (list.isEmpty())
        wait();              // Mais ici, c'est notify() sur this
      return list.removeLast();
    }
  }
}
```

# ThreadSafe Singleton

- Design Pattern ***Singleton***
  - Éviter l'allocation d'une ressource coûteuse en la réutilisant

```
public class DB {
   private DB() { ... }
   public static DB getDB() {
    if (singleton==null)
      singleton = new DB();
    return singleton;
  }
  private static DB singleton;
}
```

Problèmes/risques :
(1) Scheduling entre la création et l'assignation: 2 threads testent null en même temps: 2 objets sont créés
(2) publication de champs non initialisés

# Une bonne idée... qui finit mal

- Design Pattern ***Double-Checked Locking***
  - On peut essayer de "protéger" l'allocation, tout en limitant la nécessité de synchroniser
    - force la mise à jour
    - assure que new n'est fait qu'1 fois

```
public class DB {
   private DB() { ... }
   public static DB getDB() {
    if (singleton==null)
      synchronized(DB.class) {
        if (singleton==null)
          singleton = new DB();
      }
    return singleton;
  }
  private static DB singleton;
}
```

Reste encore le problème de la publication de champs non initialisés

# Le DCL ne marche pas

```
public class DB {
   private DB() { ... }
   public static DB getDB() {
    if (singleton==null)
      synchronized(DB.class) {
        if (singleton==null)
          singleton = new DB();
      }
    return singleton;
  }
  private static DB singleton;
}
```

- 2 threads A et B:
  - A entre dans getDB()
  - B entre dans getDB(), prend le moniteur et fait:
    - **a)** allocation par new
      **b)** appel au constructeur **DB()** (init des champs)
      **c)** affectation de la référence dans **singleton**
  - Problème: A (qui n'a pas encore pris le moniteur) peut percevoir les opérations **a) b) c)** dans le désordre.
    - Par exemple, elle peut voir **c)** (disposer de la référence sur le singleton != null) avant de voir les effets de **b)** (et donc utiliser le singleton non initialisé)

# La solution

- Utiliser le fait que les classes sont chargées de façon paresseuse (***lazy***)

```
public class DB {
    public static DB getDB() {
      return Cache.singleton;
   }
   static class Cache {
     static final DB singleton=new DB();
   }
}
```

- Le **chargement** d'une classe **utilise un verrou** et la VM garantit que la classe Cache ne sera chargée qu'une seule fois, si nécessaire.

# Atomicité et Incrémentation

- **Problème** car l'incrémentation et l'assignation sont deux opérations différentes

```
public class IDGenerator {
  private volatile long counter=0;
  public long getId() {
    return counter++;  // <==> counter=counter+1;
  }
}
```

Scheduling entre la lecture et de l'écriture

- On peut utiliser synchronized mais il y a mieux

```
public class IDGenerator {
  private final Object m = new Object();
  public volatile long counter=0;
  public long getId() {
    synchronized(m) {
      return counter++;
    }
  }
}
```

# De nombreuses classes utilitaires

- Dans java.util.concurrent depuis 1.5, 1.6
  - Facilite l'utilisation des threads
  - Permet des implémentations spécifiques aux plate-formes/processeurs
  - Pour la gestion de l'atomicité des  lectures / opérations / écriture
    - java.util.concurrent.atomics
  - Pour la gestion de la synchronisation : sémaphores, loquets à compteur, barrières cycliques, échangeurs
    - java.util.concurrent
  - Pour la gestion de la synchronisation, verrous, conditions d'attente/notification
    - java.util.concurrent.locks

# Utilitaires de java.util.concurrent.atomics

- Opérations atomiques sur différents types de variable : étend la notion de *volatile*
  - Pour chacune des différentes classes AtomicBoolean, AtomicInteger, AtomicLong, AtomicReference... on peut créer un objet atomique: `new AtomicMachin(Machin initEventuel)`:
    - Autorise des implémentations plus efficaces en fonction des possibilités offertes par les processeurs
    - Opérations atomiques inconditionnelles:
      - get(): effet mémoire de la lecture d'un champ volatile
      - set(): effet mémoire de l'écriture d'un champ volatile
      - getAndSet(): effet mémoire de lecture et d'écriture d'un champ volatile
        - Comme getAndIncrement(), getAndAdd(), etc pour les entiers

# Opérations atomiques conditionnelles

- **`boolean compareAndSet(expectedValue, updateValue)`**
  - Affecte atomiquement la valeur updateValue dans l'objet atomique si la valeur actuelle de cet objet est égale à la valeur expectedValue.
    - Retourne true si l'affectation a eu lieu, false si la valeur de l'objet atomique est différente de expectedValue au moment de l'appel.
  - Effet mémoire: lecture/écriture d'un champ volatile
- **`boolean weakCompareAndSet(expectedValue, updateValue)`**
  - Même chose, mais peut échouer (retourne false) pour une raison «inconnue» (*spurious*) : on a le droit de ré-essayer.
  - Effet mémoire: ordonnancement correct des opérations sur cette variable, mais pas de "barrière"

# Opération atomique paresseuse

- `void lazySet(value)`
  - Effet mémoire d'une écriture volatile, sauf qu'il autorise des réordonnancements avec les actions mémoires ultérieures n'ayant pas de contraintes avec les écritures non volatiles (ce qui n'est pas permit avec une écriture volatile)
  - Utilisation: mettre une valeur à null pour qu'elle soit garbage collectée

# Modification de valeur

- Si 2 threads exécutent `ai.getAndAdd(5);` sur un objet `AtomicInteger ai = new AtomicInteger(10);`
  - Alors on est assuré que les opérations ne sont pas entrelacées, qu'au final ai.get() vaut 20 et que l'une aura eu 10, l'autre 15.
  - Pas de verrou, ni de synchronisation (moins **lourd**)
- MAIS: les threads sont en concurrence et si les instructions s'entrelacent, certaines opérations peuvent échouer (affectations conditionnelles)

```
void twice(AtomicInteger ai){
  while (true) {
    int oldValue = ai.get();
    int newValue = oldValue*2;
    if (compareAndSet(oldValue,newValue))
      break;
  }
}
```

# Autres atomics spéciaux

- Atomic[ *Reference* | *Integer* | *Long* ]FieldUpdater<T,V> permettent d'utiliser des opérations "*compareAndSet*" sur les champs **volatiles** d'une classe donnée.
  - `static <T,V> AtomicReferenceFieldUpdater<T,V> newUpdater(Class<T> tclass, Class<V> vclass, String volatileFieldName)`
  - Utilisation classique: `compareAndSet(T obj,V expect,V update)`
- Atomic[ *Integer* | *Long* | *Reference* ]Array: idem sur éléments de tableaux
- AtomicMarkableReference<V> associe un booléen à une référence et AtomicStampedReference<V> associe un entier à une référence
  - `boolean compareAndSet(V expectRef, V newRef, int expectStamp, int newStamp)`

# Utilitaires de synchronisation

- `java.util.concurrent.Semaphore`
- Permet de limiter le nombre d'accès à une ressource partagée entre différentes threads, en comptant le nombre d'autorisations acquises et rendues
- `Semaphore sema = new Semaphore(MAX, true)` crée un objet sémaphore équitable (`true`) disposant de `MAX` autorisations
- `sema.acquire()` pour "décompter" de 1, `sema.release()` pour "rendre" l'autorisation

# Semaphore (acquisition d'autorisation)

- `sema.acquire()`
  - la thread courante (t) tente d'acquérir une autorisation.
    - Si c'est possible, la méthode retourne et décrémente de un le nombre d'autorisations.
    - Sinon, t bloque jusqu'à ce que:
      - Soit la thread t soit interrompue (l'exception InterruptedException est levée, tout comme si le statut d'interruption était positionné lors de l'appel à acquire())
      - Soit une autre thread exécute un release() sur ce sémaphore. T peut alors être débloquée si elle est la première en attente d'autorisation (notion d'«équité» paramétrable dans le constructeur)

# Semaphore (relâchement d'autorisation)

- `sema.release()`
  - incrémente le nombre d'autorisations. Si des threads sont en attente d'autorisation, l'une d'elles est débloquée
    - Une thread peut faire release() même si elle n'a pas fait de acquire() !
    - Différent des moniteurs

# Sémaphores

- Sémaphore *binaire*: créé avec 1 seule autorisation
  - Ressource soit disponible soit occupée
    - Peut servir de verrou d'exclusion mutuelle
    - MAIS le verou peut être «relâché» par une autre thread !
- Paramètre d'équité du constructeur: (*fair*)
    - Si false:
      - aucune garantie sur l'ordre d'acquistion des autorisations
      - Par ex: une thread peut acquérir une autorisation alors qu'une autre est bloquée dans un acquire() (*barging*)
    - Si true:
      - L'ordre d'attente FIFO est garanti (modulo l'exécution interne)

# Sémaphores (acquisition)

- Tentative d'acquisition insensible au statut d'interruption
  - void acquireUninterruptibly() : si la thread est interrompue pendant l'attente, elle continue à attendre. Au retour, son statut d'interruption est positionné.
- Tentative d'aquistion non bloquante ou bornée
  - boolean tryAcquire() [ne respecte pas l'équité] et boolean tryAcquire(long timeout, TimeUnit unit) throws InterruptedException

# Sémaphores

- Surcharge des méthodes pour plusieurs autorisations
  - acquire(int permits), release(int permits), acquireUninterruptibly(int permits)...
- Manipulation sur le nombre d'autorisations
  - int availablePermits(), int drainPermits() et void reducePermits(int reduction)
- Connaissance des threads en attente d'autorisation
  - boolean hasQueuedThreads(), int getQueueLength() Collection<Thread> getQueuedThreads()

# Loquet avec compteur

- java.util.concurrent.CountDownLatch
- Initialisé (constructeur) avec un entier
- Décrémenté par la méthode countDown()
- Une ou plusieurs threads se mettent en attente sur ce loquet par await() jusqu'à ce que le compteur "tombe" à zéro.
- Le compteur ne peut pas être réinitialisé (sinon, CyclicBarrier)
- L'appel à countDown() ne bloque pas

```
class Driver { // ...
   void main() throws InterruptedException {
     CountDownLatch startSignal = new CountDownLatch(1);
     CountDownLatch doneSignal = new CountDownLatch(N);
     for (int i = 0; i < N; ++i) // create and start threads
       new Thread(new Worker(startSignal, doneSignal)).start();
     doSomethingElse();            // don't let run yet
     startSignal.countDown();      // let all threads proceed
     doSomethingElse();
     doneSignal.await();           // wait for all to finish
   }
 }
 class Worker implements Runnable {
   private final CountDownLatch startSignal;
   private final CountDownLatch doneSignal;
   Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {
      this.startSignal = startSignal;
      this.doneSignal = doneSignal;
   }
   public void run() {
      try {
        startSignal.await();
        doWork();
        doneSignal.countDown();
      } catch (InterruptedException ex) {} // return;
   }
   void doWork() { ... }
 }
```

# Barrière cyclique java.util.concurrent.CyclicBarrier

- Même genre CountDownLatch, réinitialisable (reset())
- Si créé avec 3, chaque thread qui fait await() sur la barrière attend que 3 threads aient fait await()
- Chaque await() retourne l'ordre d'arrivée à la barrière
- Pour connaître l'état de la barrière
  - getParties() donne le nombre de threads concernées
  - getNumberWaiting() le nombre de threads en attente
- ***Broken barrier*** (tout ou rien) dans le cas où
  - Si 1 thread quitte le point de barrière prématurément
    - Interruption, problème d'exécution, reset de la barrière
  - Dans ce cas, toutes les threads en attente sur await() lèvent BrokenBarrierException
- Constructeur accepte un Runnable spécifiant du code à exécuter par la dernière thread arrivée

# Echangeur java.util.concurrent.Exchanger

- Exchanger<V> permet à 2 threads d'échanger deux valeurs d'un même type V
- `V exchange(V x) throws InterruptedException` est une méthode bloquante qui attend qu'une autre thread ait fait le même appel sur le même objet échangeur
  - À cet instant, chaque méthode retourne la valeur passée en argument par l'autre thread
  - Lève InterruptedException si la thread est interrompue
- `V exchange(V x,long timeout,TimeUnit unit) throws InterruptedException,TimeoutException`

  permet de borner l'attente

# Utilitaires de verrous

- java.util.concurrent.locks
  - Interface Lock pour une gestion des *verrous* différenciée de la notion de *moniteur* et de *bloc de synchronisation* du langage
  - Volonté de ne pas être lié à la structure de bloc
    - Prendre le verrou A, puis B, puis relâcher A, puis prendre C, puis relâcher B, etc. Impossible avec synchronized.
    - Nécessite encore plus de précautions de programmation, principalement pour assurer qu'on relâche toujours tous les verrous
  - Différentes sémantiques dans les implantations
    - Implantations principales: ReentrantLock (une seul thread accède à la ressource) et ReentrantReadWriteLock (plusieurs threads peuvent lire simultanément)

# Interface Lock

- lock() prend le verrou si disponible et sinon endort la thread
- unlock() relâche le verrou
- lockInterruptibly() peut lever InterruptedException si le statut d'interruption est positionné lors de l'appel ou si la thread est interrompue pendant qu'elle le détient
- `boolean tryLock()` et
  `boolean tryLock(long time,TimeUnit unit) throws InterruptedException`
- `Condition newCondition()` crée un objet Condition associé au verrou (voir wait/notify)
- Synchronisation de la mémoire
  - Une prise de verrou réussie et son relâchement doivent avoir le même effet sur la mémoire que l'entrée et la sortie d'un bloc synchronisé (écriture/invalidation registres et contraintes réordonnancement)

# java.util.concurrent.locks.ReentrantLock

- Verrou d'excl. mut. réentrant de même sémantique que les moniteurs implicites et synchronized
  - Le verrou est détenu par la dernière thread l'ayant pris avec succès. Elle peut le reprendre instantanément
  - isHeldByCurrentThread(), getHoldCount()
- Paramètre d'équité du constructeur (pas par défaut)
  - Celui qui attend depuis le plus longtemps est favorisé.
  - Ce n'est pas l'équité de l'ordonnancement.
  - tryLock() ne repecte pas l'équité (si verrou dispo, retourne true).
  - Globalement plus lent.
- Possibilité de consulter les détenteurs et les threads qui attendent le verrou, ou l'une de ses conditions associées

# Du bon usage des verrous

- Toujours s'assurer que le verrou sera relâché

```
class X {
  private final ReentrantLock lock=new ReentrantLock();
  // ...
  public void m() {
    lock.lock(); // block until condition holds
    try {
      // ... method body
    } finally {
      lock.unlock()
    }
  }
}
```

# Interface Condition

- java.util.concurrent.locks.Condition
  - Permet de représenter, pour n'importe quel objet verrou de type Lock, l'ensemble des méthodes habituellement associées aux moniteurs (wait(), notify() et notifyAll()) pour une condition particulière, i.e., plusieurs Conditions peuvent être associées à un Lock donné
  - Offre les moyens de
    - mettre une thread en attente de la condition: méthodes cond.await(), équivalentes à monitor.wait()
    - et de la réveiller: méthodes cond.signal(), équivalentes à monitor.notify()
    - La thread qui exécute ces méthodes doit bien sûr détenir le verrou associé à la condition

# Attente-notification des conditions: await()

- Relâche le verrou (Lock) associé à cette condition et inactive la thread jusqu'à ce que :
  - Soit une autre thread invoque la méthode signal() sur la condition et que cette thread soit celle qui soit réactivée
  - Soit une autre thread invoque la méthode signalAll() sur la condition
  - Soit qu'une autre thread interrompe cette thread (et que l'interruption de suspension soit supportée par l'implantation de await() de cette Condition, ce qui est le cas dans l'API)
  - Soit un réveil «illégitime» intervienne (*spurious wakeup*, plate-forme sous-jacente)
- Dans tous les cas, il est garanti que lorsque la méthode retourne, le verrou associé à la condition a été repris
- Si le statut d'interruption est positionné lors de l'appel à cette méthode ou si la thread est interrompue pendant son inactivité, la méthode lève une InterruptedException

# Attente-notification (suite)

- `void awaitUninterruptibly()`
  - Ne tient pas compte du statut éventuel d'interruption (qui est alors laissé)
- `long awaitNanos(long nanosTimeout) throws InterruptedException`
  - Retourne (une approximation) du nombre de nanosecondes restant avant la fin du timeout au moment où la méthode retourne, ou une valeur négative si le timeout a expiré
- `boolean await(long time,TimeUnit unit) throws InterruptedException`
  - Retourne (`awaitNanos(unit.toNanos(time)) > 0`)
- `boolean awaitUntil(Date deadline) throws InterruptedException` (pareil)
- `void signal(), void signalAll()`
  - Réactive une ou toutes les threads en attente de cette condition

# Structures de données pour concurrence

- La concurrence nécessite des collections, files d'attentes spécifiques: ex: pool de threads
- Les files d'attentes Queue<E> et Deque<E> (*Double-Ended Queue*)
- Collections **threads safe**
  - Les collections "concurrentes"
  - Les collections "classiques" synchronisées par les méthodes statiques de java.util.Collections
  - <T> Collection<T> synchronizedCollection(Collection<T> c)
    <T> Set<T> synchronizedSet(Set<T> s)
    <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)
    <T> List<T> synchronizedList(List<T> list)
    <K,V> Map<K,V> synchronizedMap(Map<K,V> m)
    <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m)
    - **Synchroniser l'accès via Iterateur!**

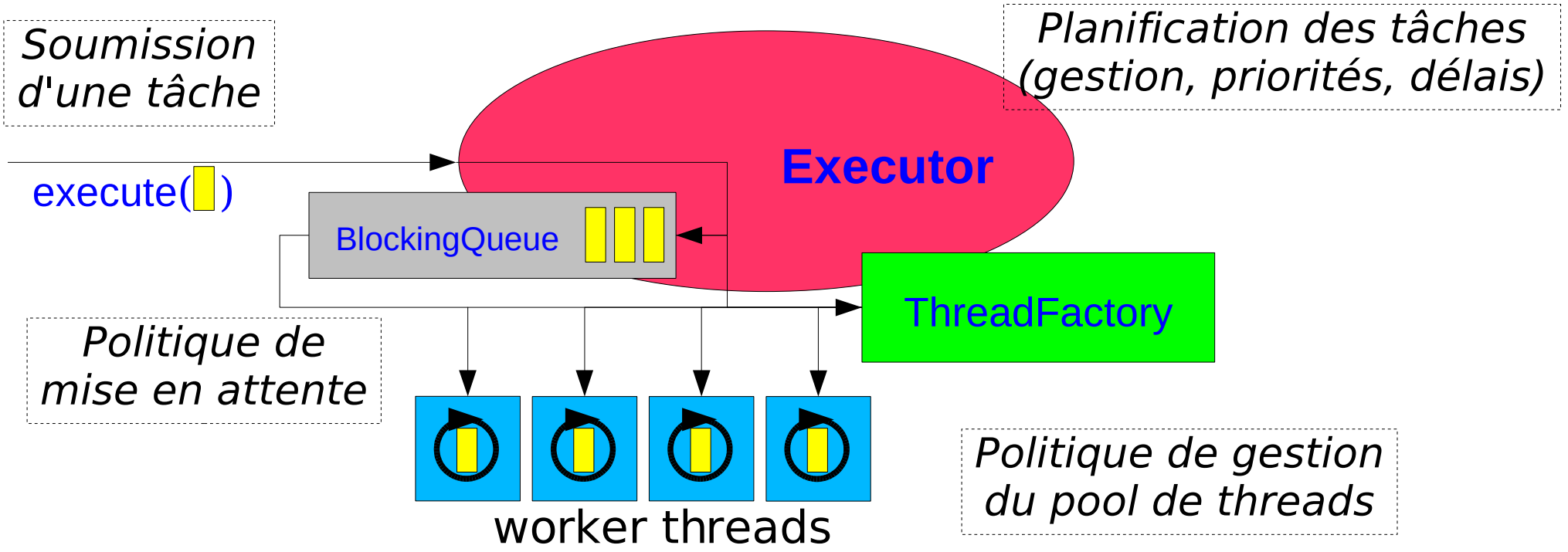
# Paramètres des pools de threads

- **Nombre de threads** qui sont créées, actives ou attendant une tâche à executer
  - getPoolSize() existantes
  - [get/set]corePoolSize() gardées dans le pool, même si inactives
    - Par défaut, elles ne sont créées qu'à la demande. On peut forcer par prestartCoreThread() ou prestartAllCoreThreads()
    - [get/set]MaximumPoolSize() maximum à ne pas dépasser
- Temps d'inactivité avant terminaison des threads du pool
  - [get/set]KeepAliveTime() si **poolSize>corePoolSize**.
  - Par défaut, Executors fournit des pool avec keepAlive de 60s

# Paramètres des pools de threads

- ThreadFactory (interface)
  - Spécifiable par le constructeur. Executors.defaultThreadFactory() crée des threads de même groupe, de même priorité normale et de statut non démon
  - Executors.privilegedThreadFactory() idem avec en plus même AccessControlContext et contextClassLoader que la thread appelante

# Architecture des Executor



# Gestion des tâches

- Lorsque une requête de planification de tâche arrive (execute()) à un Executor, il peut utiliser une file d'attente
  - Si **poolSize < corePoolSize**
    - **Créer une thread** plutôt que de mettre en file d'attente
  - Si **poolSize >= corePoolSize**,
    - **Mettre en file d'attente** plutôt que de créer une thread
  - Si la **requête ne peut pas être mise en file d'attente**
    - Une **nouvelle thread est créée** à moins que **maximumPoolSize** soit atteint
      - dans ce cas, la **requête est rejetée**

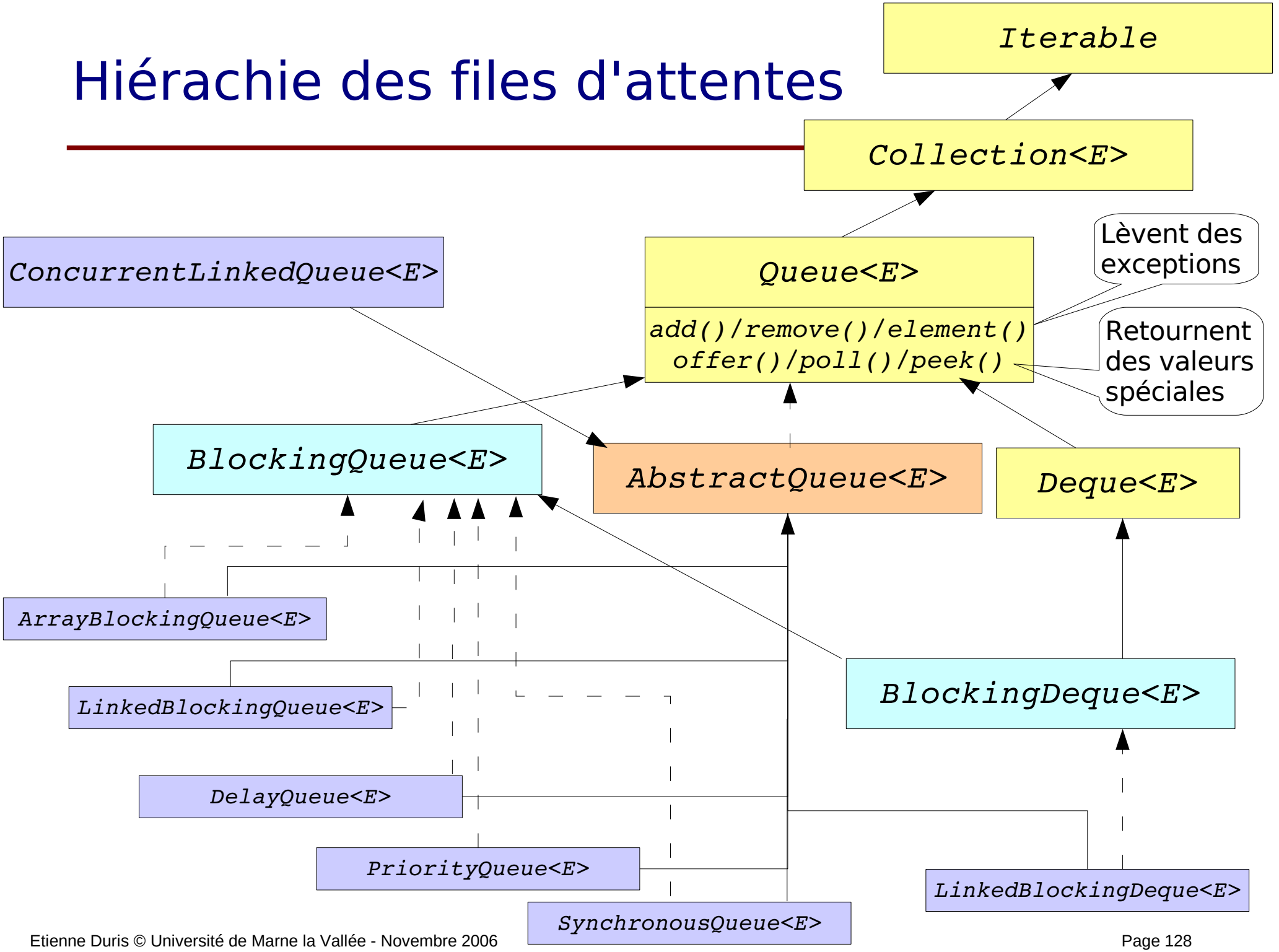
# Tâches rejetées

- Si on appelle execute() avec une tâche sur un Executor
  - qui a déjà été arrêté par shutdown() ou
  - qui utilise une taille maximale finie pour son pool de thread et pour sa file d'attente et qui est "saturé"
- Alors il invoque la méthode
  - rejectedExecution(Runnable tache,ThreadPoolExecutor this) de son RejectedExecutionHandler qui peut être une des quatre classes internes de ThreadPoolExecutor:
    - AbortPolicy: lève l'exception RejectedExecutionException (Runtime)
    - CallerRunsPolicy: la thread appelant execute() se charge de la tâche rejetée
    - DiscardPolicy: la tâche est simplement jetée, oubliée
    - DiscardOldestPolicy: si l'Executor n'est pas arrêté, la tâche en tête de file d'attente de l'Executor est jetée, et une nouvelle tentative de planification est tentée (ce qui peut provoquer à nouveau un rejet...)

# Politique de mise en attente

- Passages direct (*direct handoffs*)
  - L'attente de mise dans la file échoue si une thread n'est pas immédiatement disponible: création d'une thread
  - Ex: SynchronousQueue
- Files d'attente non bornées
  - Si les corePoolSize thread de base sont déjà créées et occupées, alors la requête est mise en attente (bien si threads indépendantes)
  - Ex: LinkedBlockingQueue
- Files d'attente bornées
  - Équilibre à trouver entre la taille de la file d'attente et la taille maximale du pool de threads
  - Ex: ArrayBlockingQueue

# Hiérachie des files d'attentes



# Les files d'attente : BlockingQueue<E>

- java.util.concurrent.BlockingQueue<E> extends java.util.Queue<E> (FIFO)
  - Opérations peuvent bloquer en attendant place/élément
  - Ne peuvent pas contenir d'élément null
  - Toutes les opérations sont **thread-safe**
  - Les méthodes d'ajout/retrait d'1 élément sont **atomiques**

| | *Throws exception* | *Special value* | *Blocks* | *Times out* |
|---|---|---|---|---|
| **Insert** | `add(e)` | `offer(e)` | `put(e)` | `offer(e, time, unit)` |
| **Remove** | `remove()` | `poll()` | `take()` | `poll(time, unit)` |
| **Examine** | `element()` | `peek()` | *not applicable* | *not applicable* |

# Accès à une BlockingQueue

- Méthodes bloquantes
  - void put(E o) throws InterruptedException
    // attend qu'il y ait de la place
  - E take() throws InterruptedException
    // attend qu'il y ait un élément
- ***Bulks*** opérations **pas atomiques**
  - Par exemple, addAll(), containsAll(), retainAll() et removeAll() peuvent échouer (lever une exception) alors que seuls certains éléments ont été "traités"
- Méthodes pour extraire les éléments d'une BQ:
  - `int drainTo(Collection<? super E> c)` (retire tous éléments disponibles)
  - `int drainTo(Collection<? super E> c, int maxElem)` `(au plus maxElem)`
  - Si exception lors d'ajout d'éléments dans c, les éléments peuvent être dans les 2 ou dans aucune des 2 collections !

# ArrayBlockingQueue et LinkedBlockingQueue

- Implémentent BlockingQueue par tableau et liste chaînée
  - ArrayBlockingQueue peut être paramétré pour être "équitable" (*fair*) ou non dans l'accès des threads mises en attente
  - LinkedBlockingQueue peut être optionnellement bornée à la création par une capacité maximale
  - Implémentent Iterable: l'itération est faite dans l'ordre
    - "***weakly consistent***" itérateur qui ne lève jamais ConcurrentModificationException. Les modifications ultérieures à la création peuvent (ou non) être reflétées par l'itérateur.
  - int remainingCapacity() retourne la capacité restante ou Integer.MAX_VALUE si la file n'est pas bornée
  - int size() donne le nombre d'éléments dans la file

# Autres files d'attente

- DelayQueue<E extends Delayed>
  - File d'attente non bornée dans laquelle ne peuvent être retirés les éléments que lorsque leur délai (getDelay()) a expiré
- SynchronousQueue<E>
  - File de capacité zéro dans laquelle chaque put() doit attendre un take() et vice versa (sorte de canal de rendez-vous)
  - Par défaut, pas «équitable», mais spécifiable à la construction
- PriorityBlockingQueue<E>
  - File d'attente non bornée qui trie ses éléments selon un ordre fixé à la construction (un Comparator<? super E> ou éventuellement l'ordre naturel).
  - Attention: l'itérateur ne respecte pas l'ordre

# Collections concurrentes spécifiques

- Offrent des méthodes dont les implantations peuvent être rendues thread-safe par des mécanismes tels que *atomics*, *locks*, etc...
- Interface ConcurrentMap<K,V> et l'implantation ConcurrentHashMap<K,V>: ajoute les méthodes
  - V putIfAbsent(K key, V value)
  - boolean remove(Object key, Object value)
  - V replace(K key, V value)
  - boolean replace(K key, V oldValue, V newValue)
- Interface ConcurrentNavigableMap<K,V> et l'implantation ConcurrentSkipList<K,V>
- Implémentation concurrente de Queue<E>
  - ConcurrentLinkedQueue<E>

# Les restes...

- Quelques bricoles que je n'ai pas réussi à mettre ailleurs...

# Les crochets d'arrêt (*Shutdown Hooks*)

- Plusieurs façons de sortir d'une exécution de JVM
  - Normales:
    - Fin de tous les processus légers non *daemon*
    - Appel de la méthode exit() de l'environnement d'exécution
  - Plus brutales:
    - Control-C
    - Levée d'erreur jamais récupérée
- Corchets d'arrêt
  - Portion de code exécutée par la JVM quand elle s'arrête
  - Sous la forme de processus légers

# Enregistrement des crochets d'arrêt

- Sur l'objet runtime courant Runtime.getRuntime()
  - addShutdownHook(...) pour enregistrer
  - removeSutdownHook(...) pour désenregistrer
  - Avec comme argument un contrôleur de processus léger, héritant de la classe Thread, dont la méthode run() de la cible spécifie du code
  - Lorsqu'elle s'arrête, la JVM appelle leur méthode start()
    - Aucune garantie sur l'ordre dans lequel ils sont exécutés
    - Chacun est exécuté dans un processus léger différent

# Exemple de crochet d'arrêt

```
import java.io.*;
import java.util.*;
public class HookExample {
  public static void main(String[] args)
                                    throws IOException {
    Thread hook = new Thread() {
      public void run() {
        try {
          out.write("Exec. aborted at"+new Date());
          out.newLine();
          out.close();
        } catch(IOException ioe) {
          ioe.printStackTrace();
        }
      }
    };
    // à suivre...
```

# Exemple de crochet d'arrêt (2)

```
// ...suite du main
    final BufferedWriter out = new BufferedWriter(
                        new FileWriter("typed.txt"));
    BufferedReader kbd = new BufferedReader(
                        new InputStreamReader(System.in));
    Runtime.getRuntime().addShutdownHook(hook);
    String line;
    while ((line = kbd.readLine()) != null) {
      if (line.equals("halt"))
      Runtime.getRuntime().halt(1);
      out.write(line);
      out.newLine();
    }
    Runtime.getRuntime().removeShutdownHook(hook);
    out.close();
    kbd.close();
  }
}
```

# Résultat

- Si le flot d'entrée est fermé normalement (Ctrl-D)
  - Le crochet est désenregistré (jamais démarré par JVM)
  - Les flots sont fermés (le contenu purgé dans le fichier)
- Si la machine est arrêtée par un Ctrl-C ou par une exception non récupérée
  - Le crochet est démarré par la JVM
  - Le fichier contient : « `Exec. aborted at...` »
- Si la JVM est arrêtée par halt() ou par un KILL
  - Le crochet n'est pas démarré
  - Potentiellement, le flot n'a pas été purgé dans le fichier