La programmation concurrente en Java

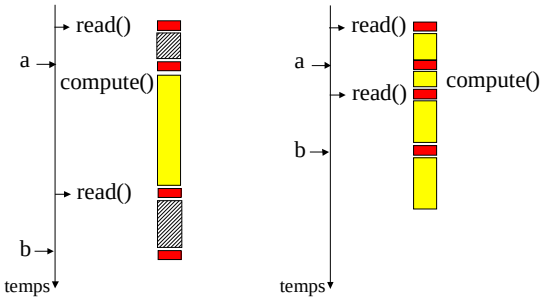
- À quoi ça sert?
  - Quelques exemples typiques.
- Quels moyens pour y parvenir?
  - La gestion de la concurrence peut être réalisée au niveau du système ou au niveau de la machine virtuelle.
- Les classes Java qui s'y rapportent.
  - Principalement **`Thread`** et **`Runnable`**
- Mise en oeuvre et problèmes spécifiques.
  - Exclusion mutuelle, synchronisation, etc.

À quoi ça sert?

- 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.

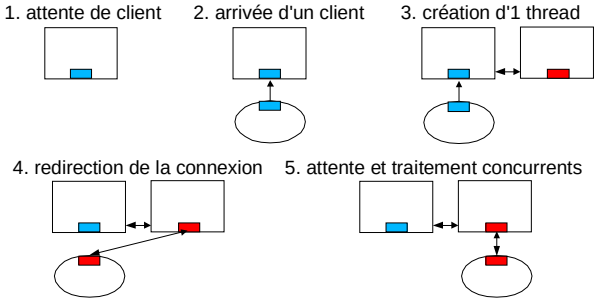
Un exemple: calculs et interactions

- Le temps d'attente d'une donnée utilisateur peut être exploité pour le calcul.



Autre exemple: client / serveur

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



Les moyens dont on dispose

- Processus du système sous-jacent
  - Déléguer la gestion de la concurrence au système
  - Assez lent, peu de mécanismes de contrôle
  - Classes **`Process`** et **`Runtime`**
- Processus légers, gérés par la machine virtuelle
  - Permet un contrôle plus fin
  - Facilite la communication entre processus légers
  - Classe **`Thread`**, qui représente un fil d'exécution
  - Classe **`Runnable`**, qui représente le code à exécuter

Les processus: commandes du système d'exploitation

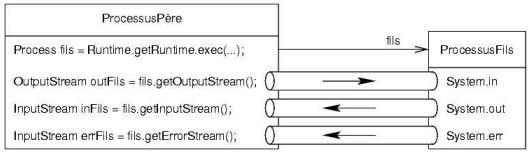
- **`java.lang.Runtime`**
  - Objet de contrôle de l'environnement d'exécution. L'objet courant peut être obtenu par la méthode statique **`Runtime.getRuntime()`**
  - D'autres méthodes: **`totalMemory(), freeMemory(), gc(), exit(), halt()...`**
  - **`exec()`** crée un nouveau processus.
- **`java.lang.Process`**
  - Objets de contrôle d'un processus, ou commande.
  - **`Runtime.getRuntime().exec("cmd")`** crée un nouveau processus et retourne un objet de cette classe.

La classe `Runtime`

- Différentes méthodes exec() : Elles créent un processus natif.
  - La plus 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`

- Retourné par les méthodes 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:
    - destroy(), getInputStream(), getOutputStream(), getErrorStream()



Les processus légers (threads): « processus » internes à la JVM

- Étant donnée une exécution de Java (une JVM)
  - un seul processus (au sens système d'exploitation)
  - 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 respecter leur séquencement.
- Deux instructions de deux processus légers distincts n'ont pas d'ordre d'exécution à respecter.

En temps normal

- Lorsque est exécutée la commande `% java Prog`
  - La JVM démarre plusieurs threads, dont, par exemple:
    - "Signal Dispatcher", "Finalizer", "Reference Handler", etc.
    - et surtout la thread "main".
    - on peut avoir ces informations en envoyant un signal QUIT au programme (Ctrl-\ sous Unix ou Ctrl-Pause sous Windows).
  - 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 (ramasse-miettes, etc).

Exemple

- Programme simple (boucle infinie dans le main())
  - Taper Crtl-\ produit l'affichage suivant (extrait):
- Full thread dump Java HotSpot(TM) Client VM (1.4.1-b21 mixed mode):

```
"Signal Dispatcher" daemon prio=1 tid=0x808c818 nid=0x33e ...
"Finalizer" daemon prio=1 tid=0x8086290 nid=0x33b ...
"Reference Handler" daemon prio=1 tid=0x80856d0 nid=0x33a in ...
"main" prio=1 tid=0x8051a20 nid=0x337 runnable [bfffd000..bfffd674]
        at Point.main(Point.java:13)
"VM Thread" prio=1 tid=0x8082518 nid=0x339 runnable
"VM Periodic Task Thread" prio=1 tid=0x808b470 nid=0x33c waiting...
"Suspend Checker Thread" prio=1 tid=0x808be38 nid=0x33d runnable
```

- Il peut y en avoir d'autres ("DestroyJavaVM", "Low Memory Detector", "CompilerThread0")

La classe `java.lang.Thread`

- Chaque instance de la classe Thread possède:
  - un nom, [get/set]Name(), identifiant
  - une priorité, [get/set]Priority(),
    - les threads de priorité plus haute sont exécutées plus 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() {...}

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 {
   // Affiche les caractéristiques du processus léger courant
   Thread t = Thread.currentThread();
   System.out.println(t);
   // Donne un nouveau nom au processus léger
   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
%
```

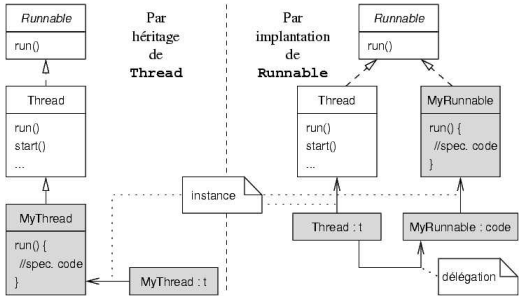
nom priorité groupe

Deux façons de spécifier le code à exécuter pour un processus léger

- Redéfinir la méthode run() de la classe Thread
  - class MyThread extends Thread {
    public 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 en Java: hériter d'une autre classe?
- un même objet Runnable exécuté par 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
  - la 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 actuellement exécutée par la JVM est retourné 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()
  - [éventuelles] pauses (laisse la main): 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

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
  - Spec. trop brutale: l'oublier
- 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, alors ce canal est fermé, le statut reste positionné et la thread reçoit une ClosedByInterruptException
- 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 était 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 interrupt (1)

```
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 + "ième exécution de " + id);
      i++;
    }
    System.out.println("Fin d'exécution du code " + id);
    // L'appel à interrupted() a réinitialisé le statut d'interruption
    System.out.println(Thread.currentThread().isInterrupted()); // Affiche: false
  }
}
```


Exemple interrupt (2)

```
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
   BufferedReader br =
     new BufferedReader(new InputStreamReader(System.in
   System.out.println("Taper le numéro de la thread à arrêter");
   switch (Integer.parseInt(br.readLine())) {
   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
10ième exécution de 1
11ième exécution de 1
12ième exécution de 1
...
```


Exemple interrupt (3)

- 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 interrupt (4)

- La méthode tempo() :

```
public void tempo() {
  try {
    Thread.sleep(Math.round(10000 * Math.random()));
  } 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
```

L'accès au processeur

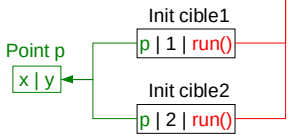
- Différents états possibles d'une thread
  - exécute son code cible (elle a accès au processeur)
  - attend l'accès au processeur (mais pourrait exécuter)
  - attend un événement particulier (pour pouvoir exécuter)
- L'exécution de la cible peut libérer le processeur
  - si elle exécute un yield() (demande 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

Les problèmes de la concurrence

- Les instructions des différentes threads peuvent, *a priori*, être exécutées dans n'importe quel ordre.
- Un seul et même espace mémoire (de la JVM) pour toutes les threads
  - Nécessite de gérer les lectures/écritures pour assurer une cohérence des données.
  - Interdire l'exécution de certaines threads pendant que l'une d'elles exécute une *portion critique.*
  - Nécessite de « synchroniser » différentes threads
- Tout cela peut provoquer des situations de famines ou d'inter-bloquage

Exemple

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




```
public class Init implements Runnable {
  Point p;
  int val;
  public Init(Point p, int val) {
    this.p = p;
    this.val = val;
  }
  public void run() {
    for(;;) {
      p.change(val,val);
      System.out.println(p);
    }
  }
  public static void main(String[] args) {
    Point p = new Point();
    Init cible1 = new Init(p,1);
    Init cible2 = new Init(p,2);
    Thread t1 = new Thread(cible1);
    Thread t2 = new Thread(cible2);
    t1.start();
    t2.start();
  }
}
```

```
% java Init
(1,1)
...
(2,2)
...
(2,1)
...
(1,2)
...
```


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).
- Impossible d'assurer qu'une 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'une thread « prend » le moniteur associé à un objet, plus aucune 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

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

- synchronized (m) {
  // bloc protégé par le moniteur m
  }
- Si on protège toute une méthode avec le moniteur associé à this, on peut utiliser le modificateur synchronized (utiliser avec précaution):

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

- Dans notre exemple, cela suffit il?
- Attention à la taille des blocs de code protégés
  => peut pénaliser l'application

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.
- Seuls les accès aux valeurs requièrent une protection par un moniteur

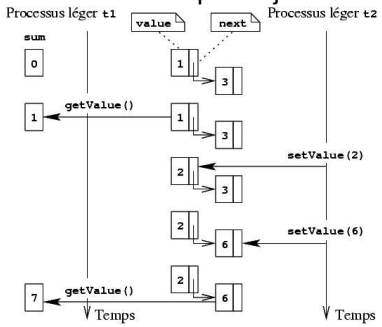
Implantation de RecList

```
public class RecList {
  private double value;
  private RecList next;
  public RecList(double value,
                 RecList next) {
    this.value = value;
    this.next = next;
  }
  public boolean contains(double x) {
    synchronized (this) {
      if (value==x)
        return true;
    }
    if (getNext()==null)
      return false;
    return getNext().contains(x);
  }
```

```
  public double sum() {
    double sum = getValue();
    if (getNext()!=null)
      sum += getNext().sum();
    return sum;
  }
  public synchronized void
                setValue(double value) {
    this.value = value;
  }
  public synchronized double getValue() {
    return value;
  }
  public RecList getNext() {
    return next;
  }
}
```

Problème

- 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 sur this
  - Le moniteur sur le premier maillon est pris et conservé jusqu'au retour de la fonction et chaque appel récursif reprend le moniteur sur le nouveau maillon.
  - n'empêche pas une modification en fin de liste entre le début et la fin du calcul de la somme.
  - PIRE: peut provoquer une situation d'inter-blocage
- Protéger les méthodes sur un seul et même moniteur partagé par tous les maillons d'une même liste.

Implantation (bis)

```
public class RecListBis {
  private double value;
  private RecListBis next;
  private Object monitor;
  public RecListBis(double value,
                    RecListBis next) {
    this.value = value;
    this.next = next;
    if (next==null)
       this.monitor = new Object();
    else this.monitor = next.monitor;
  }
  public void setValue(double value) {
    synchronized (monitor) { this.value = value; }
  }
  public double getValue() {
    synchronized (monitor) { return value; }
  }
```

```
  public double sum() {
    synchronized (monitor) {
      // Protection par ce moniteur
      // conservé récursivement
      // jusqu'à la fin du calcul de somme
      double sum = getValue();
      if (getNext()!=null)
       sum += getNext().sum();
      return sum;
    }
  }
  public boolean contains(double x) {
    synchronized (monitor) {
      if (value==x) return true;
    }
    if (getNext()==null) return false;
    return getNext().contains(x);
  }
  public RecListBis 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")) {...}
    }
    }

Synchronisation (rendez-vous)

- « Attendre » qu'un processus léger soit dans un état particulier.
  - Attendre qu'une thread t soit terminée: t.join(); accepte éventuellement une durée (en milli ou nano sec)
  - Attendre l'arrivée d'un événement particulier ou notifier l'arrivée de cet événement: wait(); et notify();
- o.wait(); exécuté par une thread donnée t1
  - requiert de détenir le moniteur o
  - suspend la thread courante t1 et libère le moniteur o
  - la thread suspendue attend (passivement) d'être réveillée par une notification sur ce même moniteur

wait() (suite)

  - Lorsque t1 reçoit une notification associée à o, elle doit à nouveau acquérir le moniteur associé à o afin de poursuivre son exécution.
- wait() peut accepter un délai d'attente en argument
  - l'attente est alors bornée par ce délai,
  - passé ce délai, la thread peut reprendre son exécution mais doit auparavant reprendre le moniteur o.
- Si une 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 une thread t2
  - par o.notify(): une seule thread en attente de notification sur o sera réveillée (choisie arbitrairement);
  - ou par o.notifyAll(): toutes les threads en attente de notification sur o seront réveillées (elles 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 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 quelqu'un d'autre

```
// 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
  - interbloquage
    plusieurs threads s'attendent mutuellement

Interbloquage

- Si on n'utilise pas les méthodes dépréciées (stop(), suspend(), resume()), le principal problème est l'interbloquage.

```
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
      }
    }
  }
}
```


Pour éviter les interbloquages

- 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 moniteur 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)

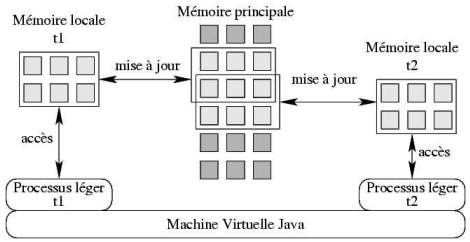
```
public class Stack {
  LinkedList list = new LinkedList();
  public synchronized void push(Object o) {
    synchronized (list) {  // Ici, le moniteur est list
      list.addLast(o);
      notify();            // Mais ici, c'est notify() sur this
    }
  }
  public synchronized Object pop() throws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


Partage des données entre threads

- Chaque thread dispose d'une mémoire locale (cache) pour stocker les « copies de travail » des variables partagées
  - réduit le temps d'accès
  - crée « plusieurs versions » de la même variable
  - problèmes de cohérence et de mise à jour
  - notion de « visibilité », depuis les autres threads, des modifications effectuées par une thread donnée

Mises à jour des variables

- Cette mise à jour est forcée
  - à la prise/libération d'un moniteur (bloc «synchronized»)
    - Entrée: force le rechargement; Sortie: force l'écriture
  - lorsqu'une thread se termine (join())



Le mot-clef *volatile*

- Utilisé comme un modificateur de champ
  - assure la cohérence entre la mémoire de travail et la mémoire principale.
  - la mise à jour est forcée pour prendre en compte les dernières modifications.
  - assure l'atomicité de la lecture et de l'écriture, y compris des champs de type double et long.
  - attention: n'assure pas l'atomicité des opérations composites, y compris l'incrémentation ou la décrémentation. En particulier, les instructions composant ces opérations peuvent s'entrelacer entre plusieurs threads concurrentes.

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 et InheritableThreadLocal 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

```
public class CodeWithLocal implements Runnable {
  public ThreadLocal local = new ThreadLocal();
  public Object shared;
  public void run() {
    String name = Thread.currentThread().getName();
    local.set(new Long(Math.round(Math.random()*100)));
    System.out.println(name + ": locale: " + local.get());
    shared = new Long(Math.round(Math.random()*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-1: locale: 96
Thread-1: partagée: 24
Thread-2: locale: 61
Thread-2: partagée: 10
Thread-1: après attente, locale: 96
Thread-1: après attente, partagée: 10
Thread-2: après attente, locale: 61
Thread-2: après attente, partagée: 10
Initial: locale: null
Initial: partagée:10
```

Transmission des variables locales

- La classe InheritableThreadLocal 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 de la thread « fille » à partir des valeurs des champs de la thread « mère », par un appel implicite à la méthode childValue() de la classe.
  - possibilité de redéfinir cette méthode dans une sous-classe de InheritableThreadLocal.

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

Java.util.concurrent

- Nouveau package jdk1.5
  - Interface Executor qui permet de « gérer » des threads
    - void execute(Runnable command)
    - Implantations: Thread-pool, IO asynchrones, tâches légères
      - Différentes fabriques d'Executor dans la classe Executors
    - Selon la classe concrète utilisée, la commande planifiée est prise en charge par un nouveau processus léger ou par un p.l. « réutilisé »ou encore par celui qui appelle la méthode execute(), et ce en concurrence ou de manière séquentielle...
  - Sous-interface ExecutorService permet de gérer plus finement la planification des commandes

ExecutorService et Future

- Utilise l'interface Callable<V>, + générale que Runnable
  - V call() throws Exception
- En + de execute(), ExecutorService définit les méthodes:
  - <T> Future<T> submit(Callable<T> task)
  - Future<?> submit(Runnable task) i.e. call() retourne null
  - <T> Future<T> submit(Runnable task, T result)
  - L'objet Future représente le résultat d'un calcul asynchrone
    - boolean cancel(boolean mayInterruptIfRunning)
    - boolean isCancelled()
    - boolean isDone()
    - V get() throws InterruptedException, ExecutionException
    - V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException

ExecutorService (suite)

- 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()), arrête d'attendre les tâches en attente, et retourne la liste des commandes n'ayant jamais commencé d'exécution
- 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, - soit le timeout ait expiré, - soit la thread courante soit interrompue
- <T> List<Future<T>> invokeAll(Collection<Callable<T>> tasks) <T> T invokeAny(Collection<Callable<T>> tasks) attendent toutes ou au moins une fin d'exécution (existent aussi avec un timout)

ScheduledExecutorService

- Cette interface ajoute
  - <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 que le délai est dépassé)
- L'énuméré TimeUnit permet de spécifier
  - MICROSECONDS, MILLISECONDS, NANOSECONDS, SECONDS
  - Offre d'autres méthodes de conversion ou d'attente

Les implantations

- ThreadPoolExecutor
  - Normalement configuré par les fabriques de Executors
    - Executors.newCachedThreadPool() pool non borné, avec réclamation automatique
    - Executors.newFixedThreadPool(int) pool de taille fixe
    - Executors.newSingleThreadExecutor() une seule thread en tâche de fond
- ScheduledThreadPoolExecutor
  - À utiliser de préférence plutôt que java.util.Timer
- FutureTask<V> implements Future<V>
  - Implémentation de base qui peut être spécialisée

Paramètres des pools de threads

- Nombre de threads qui sont créées, actives ou attendant une tâche à executer
  - [get/set]poolSize 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.new...() fournit des pool avec keepAlive de 60 sec.
- ThreadFactory
    - Spécifiable par le constructeur. Executors.defaultThreadFactory() crée des threads de même groupe, de même priorité normale et de statut non démon

Gestion des tâches

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

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

Tâches rejetées

- Si on tente de planifier une tâche par execute() dans un Executor
  - qui a déjà été arrêté par shutdown()
  - 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...)

Les files d'attente

- Interface: 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
  - Les opérations sont thread-safe (atomiques), sauf les *bulks*
    - boolean offer(E o) ajoute si possible, sinon retourne false
      - boolean offer(E o,long timeout,TimeUnit unit) throws InterruptedException
    - void put(E o) throws InterruptedException attend qu'il y ait de la place et ajoute
    - E poll() retourne la tête et l'enlève, ou null si c'est vide
      - E poll(long timeout, TimeUnit unit) throws InterruptedException
    - E peek() retourne la tête, ou null si c'est vide, mais de l'enlève pas
    - E take() throws InterruptedException attend qu'il y ait un élément et l'enlève
    - int drainTo(Collection<? super E> c) retire tous les éléments disponibles
    - int drainTo(Collection<? super E> c, int maxElem) retire au plus maxElem éléments

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, mais itère les éléments existant au moment de la création; il peut refléter les modifications ultérieures à la construction (pas garanti)
  - int remainingCapacity() retourne la capacité restante «idéale», 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

Sémaphores

- java.util.concurrent.Semaphore
  - Permet de limiter le nombre d'accès à une ressource partagée 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() : 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»)
  - 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() !

Sémaphores (suite)

- 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*:
  - 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 (fin)

- Tentative d'acquisition insensible au statut d'interruption
  - void acquireUninterruptibly() : si la thread est interrompue pendant d'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
- 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() et Collection<Thread> getQueuedThreads()

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
    - Ex: prendre le verrou A, puis B, puis relâcher A, puis prendre C, puis relâcher B, etc. Impossible avec les blocs de synchronisation.
    - Nécessite encore plus de précautions de programmation, principalement pour assurer qu'on relâche toujours tous les verrous
  - Permettre différentes sémantiques dans les implantations
    - Implantations principales: ReentrantLock (une seul thread accède à la ressource) et ReentrantReadWriteLock (plusieurs threads peuvent lire simultanément)

Interface Lock

- Méthodes déclarées
  - lock() prend le verrou s'il est 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
- 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 (synchronisation mémoire cache / mémoire principale)

java.util.concurrent.locks.ReentrantLock

- Verrou d'exclusion mutuelle réentrant de même sémantique que les moniteurs implicites et l'instruction synchronized
  - Le verrou est détenu par la dernière thread l'ayant pris avec succès. Si elle le détient, elle peut le reprendre instantanément (réentrant)
  - isHeldByCurrentThread(), getHoldCount()
- Paramètre d'équité dans le constructeur (pas par défaut)
  - La thread qui attend depuis le plus longtemps est favorisée.
  - 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)
    - Soit un réveil «illégitime» intervienne (*spurious wakeup* , plateforme sousjacente)
  - 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

java.util.concurrent.atomics

- Package permettant des opérations atomiques sur plusieurs types de variable : étend la notion de *volatile*
  - Pour chacune des différentes classes AtomicBoolean, AtomicInteger, AtomicReference, etc. on crée un objet atomique: new Atomic*Machin*(*Machin* initEventuel) et on peut y accéder, par exemple:
  - opérations atomiques get(), getAndSet(), etc...
  - 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.
  - boolean weakCompareAndSet(expectedValue, updateValue)
    - Même chose, mais peut échouer (retourne false) pour une raison «inconnue» (*spurious*) : on a le droit de ré-essayer.

Échange de données

- Si deux threads exécutent en concurrence le code
  - ai.getAndAdd(5); sur un objet AtomicInteger créé comme ceci: 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.
- Il n'y a pas de verrou (ni de synchronisation):
  - Les threads sont en concurrence et si les instructions s'entrelacent, certaines opérations peuvent échouer (affectations conditionnelles)
  - ```
    void twiceThePreviousValue(AtomicInteger ai) {
        while (true) {
            int oldValue = ai.get(); int newValue = oldValue*2;
            if (compareAndSet(oldValue,newValue))
                break;
        }
    }
    ```


Effets sur la mémoire des Atomics

- Les effets des accès et mises à jour sur les objets atomiques suivent les règles des *volatiles*:
  - get() : même effet que lire un volatile
  - set() : même effet que l'affectation d'un volatile
  - weakCompareAndSet() : lecture atomique et écriture conditionnelle, ordonnées relativement aux autres opérations mémoires sur cette variable, mais sans les critères propres aux variables volatiles
  - compareAndSet() et autres *read-and-update* : mêmes effets que lecture et écriture d'une variable volatile
    ```