Les threads en java

http://cui.unige.ch/~billard/systemeII/cours1.html
Processus

http://www.eli.sdsu.edu/courses/fall98/cs596/notes/index.html
Pour la partie sur les threads. Attention quelques exemples ne passeront pas la compilation à cause de la version de jdk.

Processus

Chaque application (emacs, word, etc.) exécutée sur un ordinateur lui est associée un processus représentant l'activité de cette application. À ce processus est associé un ensemble de ressources propres à lui comme l'espace mémoire, le temps CPU etc. Ces ressources seront dédiées à l'exécution des instructions du programme associé à l'application.
Les processus coûtent chers au lancement (l'espace mémoire à calculer, les variables locales au processus à définir etc.).

Multitâche

C'est la possibilité que le système d'exploitation a, pour faire fonctionner plusieurs programmes en même temps et cela sans conflit. Par ex.: écouter de la music et lire son courrier en même temps.

Un système monoprocesseur (et non pas monoprocessus, ne pas confondre processus et processeur), simule le multitâche en attribuant un temps (un quantum ou temps déterminé à l'avance) de traitement pour chaque processus (description très simpliste du rôle du monoprocesseur). Un système multiprocesseurs peut exécuter les différents processus sur les différents processeurs du système. Généralement, un programme est livré avec un tel système pour permettre de distribuer de manière efficace les processus sur les processeurs de la machine.

Un programme est dit multitâche s'il est capable de lancer plusieurs parties de son code en même temps. À chaque partie du code sera associé un processus (sous-processus) pour permettre l'exécution en parallèle.

- La possibilité de lire l'aide en ligne et d'imprimer à la fois, sous word. Deux sous-processus, d'un processus qui est associé à l'application word.

- Un serveur réseau (processus) et le traitement de requêtes (sous-processus associé à chaque requête). Nous aimerions que le traitement de ses requêtes soit fait de manière optimale de telle manière à pouvoir servir toutes les requêtes dans un espace de temps raisonnable.

Comme chaque processus coûte cher au lancement, sans oublier qu'il va avoir son propre espace mémoire de ce fait la commutation (passage d'un processus à un autre) ainsi que la communication inter-processus (échange d'informations) seront lourdes à gérer.

Peut-on avoir un sous-processus, qui permettrait de lancer une partie du code d'une application sans qu'il soit onéreux? La réponse est venue avec les threads qui sont appelés aussi "processus léger".

Threads

Un thread[1] est un sous-ensemble d'un processus, partageant son espace mémoire et ses variables. De ce fait, les coûts associés suite à son lancement sont réduits, donc plus rapide. En plus de cela, à chaque thread est associé des unités propres à lui, comme sa pile (pour gérer les instructions à exécuter par le thread), le masque de signaux (les signaux que le thread doit répondre), sa priorité d'exécution (dans la file d'attente), des données privées etc. Il faut ajouter à cela, les threads peuvent être exécutés en parallèle par un système multitâche.

Cependant le partage de la mémoire et les variables du processus entraînent un certain nombre de problèmes quand par exemple il y a un accès partagé à une ressource. Deux agents (threads) voulant effectuer des réservations de places d'avion (ressource: nombre de places disponibles dans un avion). On protège l'accès à cette ressource dès qu'un thread est entrain de réaliser une écriture (réaliser une réservation) sur cette ressource.

[1] Un thread n'est pas un objet! C'est un sous-processus qui exécute une série d'instructions d'un programme donné.

Les threads et Java

Création

Pour créer un thread nous avons besoin de décrire le code qu'il doit exécuter et le démarrer par la suite:

- Un thread doit obligatoirement implanter l'interface Runnable

http://java.sun.com/j2se/1.4/docs/api/java/lang/Runnable.html

```
public interface Runnable {
      void run();
}
```

La méthode run doit contenir le code à exécuter par le thread.

- Créer un contrôleur de thread qui permet de démarrer un thread. Ceci est réalisable avec l'une des deux approches suivantes:

classe qui dérive de java.lang.thread	**classe qui implante l'interface Runnable**

`class ThreadTest extends Thread {`
`  public void  run() {`
`  // etc.`
`  }`
`  public ThreadTest(…) {`
`  // const. avec/sans args.`
`  }`
`}`
	`class ThreadTest implements Runnable`
	`{`
	`  public void  run() {`
	`  // etc.`
	`  }`
	`  public ThreadTest(…) {`
	`  // const. avec/sans args.`
	`  }`
	`}`

`ThreadTest MonThread = new ThreadTest(…);`	`ThreadTest test = new ThreadTest(…);` `Thread t1 = new Thread(test);`

Comme en Java la notion d'héritage multiple est inexistante, dans le cas où l'héritage simple est déjà pris (le cas des applets par exemple) vous devez implanter l'interface Runnable:

```
public class MonApplet extends Applet implements Runnable { //etc.}
```

- Lancer le thread

La méthode `start()` sert à lancer le thread. Vous devez l'utiliser via le contrôleur de thread, i.e.:

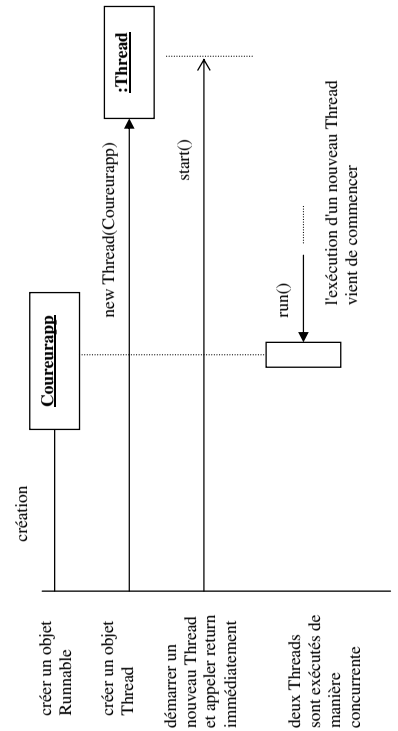
```
MonThread.start();
t1.start();
```


classe qui dérive de java.lang.thread

```
class Coureur extends Thread {

    String nom_coureur;
    /*
      constructeur de la classe
    */

    public Coureur (String str) {
      nom_coureur = str;
    }
    /*
      on redéfinit run() pour décrire le comportement d'un coureur
      quelconque
    */

    public void run() {
      /*
        la valeur i est incrémentée lors du passage à la ième centaine
        de mètres
      */

      for (int i =1; i<=10; i++) {
          System.out.println(i*100 + " m  par " + nom_coureur);
          try {
            /*
              pause aléatoire de 1 seconde maximum qui simule le
              temps mis par chaque coureur pour franchir 100 m
             */
            sleep((int)(Math.random() * 1000));
          } catch (InterruptedException e) {}
      }

      System.out.println( nom_coureur + " est arrive  ! ");
    }
}

// classe pour tester la classe Coureur
class Course {
    public static void main (String[] args) {

      // Il s'agit d'une classe de coureurs
      System.out.println("Passage aux  : ");
      Coureur  j = new Coureur ("Jean");
      Coureur  p = new Coureur ("Paul");

      // On lance les deux coureurs.
      j.start();
      p.start();
    }
}
```

classe qui implante l'interface Runnable

```
class Coureurapp implements Runnable {

    String nom_coureur;
    /*
      constructeur de la classe
    */

    public Coureurapp (String str) {
      nom_coureur = str;
    }
    /*
      on redéfinit run() pour décrire le comportement d'un coureur
      quelconque
    */

    public void run() {
      /*
        la valeur i est incrémentée lors du passage à la ième centaine
        de mètres
      */

      for (int i =1; i<=10; i++) {
          System.out.println(i*100 + " m  par " + nom_coureur);
          try {
            /*
              pause aléatoire de 1 seconde maximum qui simule le
              temps mis par chaque coureur pour franchir 100 m
             */
            Thread.sleep((int)(Math.random() * 1000));
          } catch (InterruptedException e) {}
      }

      System.out.println( nom_coureur + " est arrive  ! ");
    }
}

// classe pour tester la classe Coureur
class Courseapp {
    public static void main (String[] args) {

      // Il s'agit d'une classe de coureurs
      System.out.println("Passage aux  : ");
      Coureurapp  j = new Coureurapp ("Jean");
      Thread tj = new Thread(j);

      Coureurapp  p = new Coureurapp ("Paul");
      Thread tp = new Thread(p);

      // On lance les deux coureurs.
      tj.start();
      tp.start();
    }
}
```

La génération dynamique de Threads

La classe Thread

Nous allons présenter quelques méthodes de la classe Thread, l'ensemble de la classe est décrit sur ce lien:

http://java.sun.com/j2se/1.4/docs/api/java/lang/Thread.html

Constructeurs

public Thread(); // crée un nouveau Thread dont le nom est généré automatiquement (aléatoirement).

public Thread(Runnable target); // target est le nom de l'objet dont la méthode run est utilisée pour lancer le Thread.

public Thread(Runnable target, String name); // on précise l'objet et le nom du Thread.

public Thread(String name); // on précise le nom du Thread.

Il y a aussi des constructeurs de groupe de Threads que nous allons voir plus tard dans le cours.

Méthodes

void destroy(); // détruit le Thread courrent sans faire le ménage.

String getName(); // retourne le nom du Thread.

int getPriority(); // retourne la priorité du Thread.

void interrupt(); // interrompt le Thread.

static boolean interrupted(); // teste si le Thread courrent a été interrompu.

void join(); ou void join(long millis); ou void join(long millis, int nanos); // attendre la mort du Thread, ou après un millis de millisecondes, ou millisecondes plus nanosecondes.

void resume(); // redémarrer le Thread.

void run(); // La méthode contenant le code à exécuter par le Thread.

void setPriority(int newPriority); // changer la priorité du Thread.

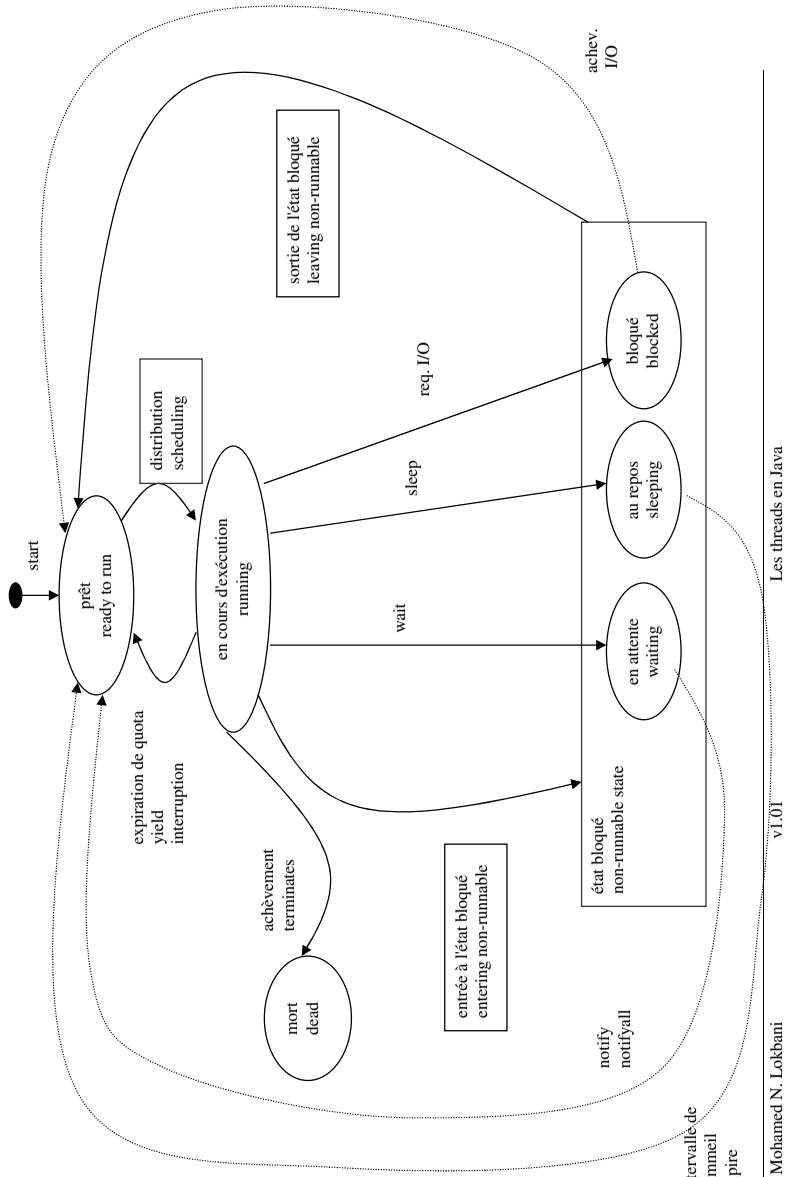
static void sleep(long millis); ou static void sleep(long millis, int nanos); // mettre en veille le Thread pendant millis millisecondes ou millisecondes plus nanosecondes.

void start(); // démarrer un Thread.

String toString(); // nom du thread, priorité, et le groupe à qui il appartient.

et bien d'autres méthodes …

Le cycle de vie d'un Thread



Un thread peut passer par plusieurs états lors de son cycle de vie:

- en cours d'exécution: le processeur est entrain d'exécuter le thread. Le mécanisme de gestion de threads (ou l'ordonnanceur) décide le thread qui doit être dans l'état d'exécution. Cette exécution sera fonction du temps cpu alloué par le système. Dès que ce temps est écoulé le thread attend que le gestionnaire le rappelle avec un nouveau quantum de temps. Un thread se retrouve dans cet état suite à l'appel de yield (pour passer la main et donner ainsi donc la chance à un autre thread afin de s'exécuter) ou bien suite à une interruption.

- état bloqué: en fonction de l'action sur le thread, ce dernier passe de l'état "en cours d'exécution" vers l'état bloqué. Il faudra une action "spéciale" pour sortir de cet état. On distingue trois possibilités pour arriver à cet état:

- état en attente: un appel à la méthode wait de la classe Object met le thread en attente. Il faudra un appel à la méthode notify (ou notifyAll) pour quitter cet état et revenir à l'état prêt.

- état au repos: un appel à la méthode static sleep permet d'endormir pendant un certain temps un thread. Ce dernier se réveille et se met à l'état prêt après l'écoulement du temps de sommeil.

- état bloqué: par exemple quand le thread est en attente d'une ressource entrée sortie (une disquette par exemple) ou bien en attente d'un moniteur d'objet (ou sémaphore: contrôle l'autorisation d'exécution d'un code synchronisé). Pour sortir de cet état, il faut que l'opération soit terminée avant de passer à l'état "prêt".

- prêt: la création du thread nécessite l'attribution de certaines ressources avant qu'il ne soit éligible pour l'exécution. Un appel à la méthode start ne signifie pas que le thread sera exécuté sur le champ, mais plutôt qu'il est prêt pour être exécuté par le système d'exploitation. Ce dernier devra allouer un temps machine (cpu) pour l'exécution de ce thread. Un thread quittant l'état bloqué doit impérativement passer par l'état "prêt" et attendre que le mécanisme de gestion de threads lui attribue un temps cpu.

- mort: un thread est dans cet état suite à la fin de l'exécution de la méthode run lui étant associée, ou bien après son arrêt suite à l'appel de la méthode stop().

L'utilisation des méthodes stop(), suspend(0) et resume(0) est fortement déconseillée à cause des situations de blocage qu'elles peuvent provoquer à cause de problèmes de communication entre les threads. Une description détaillée de ces problèmes se trouve sur cette page:

http://java.sun.com/j2se/1.4/docs/guide/misc/threadPrimitiveDeprecation.html
Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?

Description des différentes méthodes du cycle de vie d'un Thread

sleep: permet donc d'en dormir un thread un certain temps. Voir les précédents exemples pour son utilisation dans les deux cas de contrôleurs de thread. Pour utiliser sleep, il faudra capturer l'exception InterruptedException.

via une implantation de la classe Runnable

```
Thread.sleep((int)(Math.random() * 1000));
```

sinon

```
sleep((int)(Math.random() * 1000));
```

vu que la classe dérive directement de Thread.

yield: met le thread courant dans l'état prêt pour permettre à un autre thread de même priorité ou priorité supérieure de s'exécuter.

```
class  YieldThread  extends Thread {
   public  void run() {
      for  ( int  count = 0;  count < 4; count++) {
         System.out.println( count + "From: " + getName() );
         yield();
      }
   }
}
class  TestYield {
   public static void main( String[]  args )  {
      YieldThread  first      = new YieldThread();
      YieldThread  second  = new YieldThread();
      first.start();
      second.start();
      System.out.println( "End" );
   }
}
```

sortie:

```
End
0From: Thread-0
0From: Thread-1
1From: Thread-0
1From: Thread-1
2From: Thread-0
2From: Thread-1
3From: Thread-0
3From: Thread-1
```

start: exécute donc la méthode run associé au thread. Quand un thread est lancé, vous ne pouvez pas le relancer une 2e fois.

```
class  SimpleThread  extends Thread {
   public void run() {
      System.out.println(  "I ran" );
   }
}
public class Restart  {
   public static void main( String args[] ) throws Exception {
      SimpleThread onceOnly = new SimpleThread();
      onceOnly.setPriority( 6 );
      onceOnly.start();
      System.out.println( "Try restart");
      onceOnly.start();
      System.out.println( "The End");
   }
}
```

I ran
Try restart
The End

interrompre un thread: 3 méthodes de la classe Thread interviennent à ce niveau:
void interrupt(); pour interrompre un thread
boolean isInterrupted(); true s'il a été interrompu sinon false.
static boolean interrupted();true s'il a été interrompu sinon false et remet le volet (flag) interruption à son état d'origine.

```
class  NiceThread  extends Thread {
   public void run() {
      while ( !isInterrupted() ) {
         System.out.println(  "From: " + getName()  );
      }
      System.out.println( "Clean up operations" );
   }
}


public class InterruptTest {
   public static void main(String args[]) throws InterruptedException{
      NiceThread  missManners  = new NiceThread();
      missManners.setPriority( 2 );
      missManners.start();
      Thread.currentThread().sleep( 2 ); // Let other thread run
      missManners.interrupt();
   }
}
```

From: Thread-0
From: Thread-0
.....
From: Thread-0
Clean up operations

À signaler que l'attente d'entrée sortie n'est pas interrompue par Interrupt.

Les méthodes wait & notify seront étudiées en même temps que le principe de synchronisation.

join: est utilisé pour mentionner au thread d'attendre la fin d'un autre thread.

public final synchronized void join(long millis) throws InterruptedException
public final synchronized void join(long millis, int nanos) throws InterruptedException
public final void join() throws InterruptedException

La dernière variante attend indéfiniment jusqu'à la fin du thread sur lequel cette méthode s'applique.

```
Coureur  j = new Coureur ("Jean");
Coureur  p = new Coureur ("Paul");

// On lance les deux coureurs.
j.start();
p.start();
try { j.join(); p.join(); }
catch (InterruptedException e) {};
Coureur k = new Coureur ("toto");
k.start() ;
```

Ici toto va se lancer après que Jean & Paul terminent leur course.

La synchronisation

Nous avons mentionné que les Threads partagent les mêmes ressources (mémoire, variables du processus etc.). Dans certaines situations, il est nécessaire de synchroniser les threads pour obtenir un résultat cohérent.

Prenez l'exemple d'un avion où il reste une seule place de disponible et deux clients se présentant à deux différents guichets. Si la place est proposée au premier client et que ce dernier prenne tout son temps pour réfléchir, nous n'allons pas attribuer cette place au second client qui en a fait la demande. De ce fait, nous synchronisons l'accès à la méthode de réservation de telle manière à ne pas attribuer la même place aux deux voyageurs ou bien au second voyageur avant que le premier ne prenne sa décision de la prendre ou pas.

Monitor (ou sémaphore)
Le moniteur est utilisé pour synchroniser l'accès à une ressource partagée. Cette ressource peut-être un segment d'un code donné. Un thread accède à cette ressource par l'intermédiaire de ce moniteur. Ce moniteur est attribué à un seul thread à la fois (comme dans un relais 4x100m où un seul coureur tient le témoin dans sa main pour le passer au coureur suivant dès qu'il a terminé de courir sa distance). Pendant que le thread exécute la ressource partagée aucun autre thread ne peut y accéder. Le thread libère le monitor dès qu'il a terminé l'exécution du code synchronisé, ou bien s'il a fait appel à la méthode wait de l'objet.

Il faudra faire attention aux méthodes statiques qui sont associées aux classes. Synchroniser de telle méthode revient a bloqué le monitor de la classe, d'où problèmes de performances (vous bloquez l'accès à toute la classe).

On peut synchroniser une méthode ou une série d'instructions, comme suit:

```
public class SynchTest  {
   public static void main( String args[] ) throws Exception {
      LockTest aLock = new LockTest();
      TryLock tester = new TryLock( aLock );
      tester.start();

      synchronized ( aLock ) {
         System.out.println( "In Block");
         Thread.currentThread().sleep( 5000);
         System.out.println( "End Block");
      }
   }
}
class TryLock extends Thread {
   private LockTest myLock;

   public TryLock( LockTest aLock ) {
      myLock = aLock;
   }
   public void run()   {
      System.out.println( "Start run");
      myLock.enter();
      System.out.println( "End run");
   }
}
class LockTest {
   public synchronized void enter() {
      System.out.println( "In enter");
   }
}
```

```
In Block
Start run
End Block
In enter
End run
```

wait & notify

Lorsqu'un programme est multi-tâche, un thread ne peut continuer son exécution que si une condition est remplie et cette dernière est dépendante d'un autre thread. Pour que le premier thread sache que le second a modifié l'état de sa condition, il devra le tester indéfiniment, solution très onéreuse. On peut contourner cela, en utilisant le wait & notify. Le premier thread passe de l'état "en cours d'exécution" à l'état "d'attente" au même moment ou il lâche le monitor sur l'objet (ou la méthode), l'opération se fait en un temps "atomic" pour permettre à un autre thread de prendre le monitor et d'exécuter le code synchronisé. Dès que le thread modifie l'état de la condition, il appelle la méthode notify pour informer le thread en question et le fait passer donc de l'état d'attente à l'état prêt.

```
public final void wait(timeout) throws InterruptedException
```

fait que le thread attend jusqu'à notify ou que le timeout (millisecondes) a expiré.

Throws:

IllegalMonitorStateException : le thread n'est pas le propriétaire du monitor.

InterruptedException : un autre thread a provoqué son arrêt.

Un thread détient le monitor s'il exécute une méthode synchronisée, un bloc de code synchronisé, ou une méthode statique synchronisée d'une classe.

```
public final void wait(timeout, nanos) throws InterruptedException
public final void wait() throws InterruptedException

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

informe tous les threads en attente d'un changement d'une condition donnée.

```
synchronized void waitingMethod()
   {
   while ( ! condition )
      wait();
      // faire le traitement nécessaire si la condition est vraie.
   }

synchronized void changeMethod()
   {
   // Changer quelques valeurs utilisées dans la condition
   notify();
   }
```



```
class Depot extends Thread {
      private int nbJetons = 0;
      public synchronized void donnerUnJeton() {
            try {
                  while (nbJetons == 0) {
                        wait();
                  }
                  nbJetons--;
             } catch (InterruptedException e) {
             }
      }
}

public synchronized void recevoir(int n) {
      nbJetons += n;
      notifyAll();
}
```

Niveau de priorités et Threads

La priorité est bornée entre 0 & 10. La valeur par défaut est 5.

static int MAX_PRIORITY : priorité Max qu'un thread peut avoir.

static int MIN_PRIORITY : priorité Min qu'un thread peut avoir.

static int NORM_PRIORITY : priorité par défaut affectée à un thread (la valeur de 5)

int getPriority(); retourne la priorité du thread

void setPriority(int newPriority); modifie la priorité d'un thread.

```
// On lance les deux coureurs.
tj.start();
System.out.println("priorité de tj: "+ tj.getPriority()); // 5

tp.setPriority(9);
System.out.println("priorité de tp: "+ tp.getPriority()); // 9
```

ThreadGroup

C'est un ensemble de threads qui peut représenter lui aussi un threadgroup. Chaque thread donc appartient a un ensemble. Le premier threadgroup d'un programme java est représenté par "main".

la méthode tostring affiche pour le thread coureur de l'exemple course:

```
[Jean,5,main]
```

Jean étant le nom du thread, 5 sa priorité, et main son groupe de threads.

Pour créer un thread d'un groupe de threads, on crée d'abord le groupe de threads:

```
ThreadGroup monGroupe = new ThreadGroup("GroupeMaitre") ;
```

puis on utilise un constructeur qui prend en argument un groupe de threads pour créer le thread en question:

```
Thread t = new Thread(monGroupe) ;
```

Le groupe de thread permet par exemple de faciliter la gestion des threads (ils appartiennent au même groupe, plus facilement à savoir qui est qui et qui fait quoi).

Dans la classe Thread plusieurs méthodes traitent des groupes de threads, comme par exemple:

static int activeCount(); : retourne le nombre de thread actives dans le groupe de threads courrent.
ThreadGroup getThreadGroup(); : retourne le thread groupe à qui ce thread appartient.

etc.

```
class ListeDeThreads {
      void listeDesThreadsCourants() {
            ThreadGroup                  groupeCourant                  =
            Thread.currentThread().getThreadGroup();
            int nbreThreads;
            Thread[] listeDeThreads;

            nbreThreads = groupCourant.activeCount();
            listeDeThreads = new Thread[nbreThreads];
            GroupCourant.enumerate(listeDeThreads);
            for (int i = 0; i < numThreads; i++) {
                  System.out.println("Thread numéro " + i + " = " +
            listeDeThreads[i].getName());
            }
      }
}
```

à titre d'exercice implanter cette classe dans le fichier Course.java