Threads

Rappels - Définitions

un Process est un programme qui “tourne” et dispose de ses propres ressources mémoire.
OS multi-tâche $\Rightarrow$ plusieurs process en concurrence

un Thread est un flux séquentiel de contrôle à l’intérieur d’un processus
il peut y avoir plusieurs threads en concurrence pour un process, ils partagent toute ou partie des ressources mémoire du process.

Exemples :

- ▷ OS
- ▷ interface graphique
- ▷ vie artificielle
- ▷ robots pilotés par un serveur
- ▷ etc.

Java et Threads

- ▷ La gestion du multi-threading est fournie par la machine Java et l'API
  ↪ JAVA est multi-plateforme

- ▷ Dans une JVM (= 1 process JAVA) :

  → tous les threads partagent le même tas (heap)
  dans le tas sont stockées : les variables d'instances

  → tous les threads partagent la même zone de méthodes
  dans la zone de méthodes sont stockées : les variables de classe

  → chaque thread possède sa propre pile
  dans la pile sont stockés : variables locales, les paramètres des méthodes et les valeurs de retour

Mettre en place des Threads
et
Prendre en compte des accès concurrents ("*Thread safety*")

`java.lang.Thread`

- Créer une classe héritant de `Thread`
- Surcharger la méthode `Thread.run()`
- Créer une instance de la classe
- Appeler la méthode `start()` sur cette instance

```
package essais.thread;

public class TestThread extends Thread {
   private static int cpt = 0;
   private int index = cpt++;

   public void run() {
      while(true) {
         System.out.println("thread "+index+" actif");
      }
   }

   public static void main(String[] args) {
      for(int i = 0; i < 5; i++) {
         new TestThread().start();
      }
   }
} // TestThread
```

```
+-----------
| > java essais.thread.TestThread
| thread 4 actif
| thread 0 actif
| thread 1 actif
| thread 2 actif
| thread 3 actif
| thread 4 actif
| thread 0 actif
| thread 1 actif ...
```

`java.lang.Runnable`

problème si la classe hérite déjà d'une autre... (`Applet`, `Frame`, `Agent`, ...)

dans ce casimplémenter l'interface `Runnable`

- Créer une classe implémentant l'interface `Runnable`
- Implémenter la méthode `Thread.run()`
- Créer une instance de la classe
- Créer un objet `Thread` à partir de cette instance
- Appeler la méthode `start()` sur cet objet `Thread`

```
package essais.thread;
class SuperClasse {}
public class TestRunnable extends SuperClasse  // héritage possible
        implements Runnable {

   private static int cpt = 0;
   private int index = cpt++;
   public void run() {
      while(true) { System.out.println("runnable "+index+" actif"); }
   }
   public void go() { new Thread(this).start(); }

   public static void main(String[] args) {
      TestRunnable tr = new TestRunnable();
      Thread t = new Thread(tr);
      t.start();
      for(int i = 0; i < 5; i++) {
         new TestRunnable().go();
      }
   }
} // TestRunnable
```

```
+-----------------
| >java essais.thread.TestRunnable
| runnable 5 actif
| runnable 0 actif
| runnable 1 actif
| runnable 2 actif
| runnable 3 actif
| runnable 4 actif
| runnable 5 actif
| runnable 0 actif
| runnable 1 actif
```

Autres méthodes

sleep(long) (`static`)

> "*Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. The thread does not lose ownership of any monitors.*"

yield() (`static`)

> "*Causes the currently executing thread object to temporarily pause and allow other threads to execute.*"

isAlive()

Bof :

setPriority(int), getPriority()

Dépréciation

stop

suspend

resume

Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?
(`.../jdk1.2/docs/guide/misc/threadPrimitiveDeprecation.html`)

Problème : **UNSAFE**

Unsafe : accès concurrent à un même objet

Lors de l'exécution d'une méthode par un thread, l'état de l'objet peut être temporairement corrompu, il ne faudrait pas qu'un autre thread y accède à ce moment...
... malheureusement cela arrive.

- ▷ conflit écriture/écriture : l'objet résultant du conflit est dans un état invalide
- ▷ conflit écriture/lecture : l'objet est lu dans un état temporaire invalide

```
public class MultipleExclusiveChoice {
   private int currentChoice = 0;
   private boolean choice1 = true;
   private boolean choice2 = false;
   public void setChoice(boolean c1, boolean c2) {
      this.choice1 = c1; this.choice2 = c2;
   }
   public boolean validState() { return this.choice1 ^ this.choice2; } // XOR
   public void flush() { System.out.println(this.choice1+" "+this.choice2); }
}
public class ChoiceThread extends Thread {
   private MultipleExclusiveChoice mec;
   private int i;
   public ChoiceThread(MultipleExclusiveChoice mec, int i) {
      this.mec = mec; this.i = i; }
   public void run() {
      System.out.println("thread "+i+" started");
      while (true) {
         if (i==1) {
            mec.setChoice(true, false);
            System.out.println("1");
         }
         else {
            mec.setChoice(false, true);
            System.out.println("2");
         }
      }
   }
}
```

```
public class DisplayThread extends Thread {
   private MultipleExclusiveChoice mec;
   public DisplayThread(MultipleExclusiveChoice mec) {
      this.mec = mec;
   }
   public void run() {
      while (mec.validState()) { } // boucle tq valide
      System.out.println("invalid state"); mec.flush();
      System.exit(1); // carrément
   }
}
public class CorruptionThread {
   public static void main(String[] args) {
      MultipleExclusiveChoice mec = new MultipleExclusiveChoice();
      ChoiceThread t1 = new ChoiceThread(mec,1);
      ChoiceThread t2 = new ChoiceThread(mec,2);
      DisplayThread dt = new DisplayThread(mec);
      t1.start(); t2.start();
      dt.start();
   }
} // CorruptionThread
```

```
+-----------------
| thread 1 started
| 1
| thread 2 started
| 1
| 2
| invalid state
| 1
| 2
| false false
+-----------------
```

Synchronisation

Nécessité de protéger contre les accès concurrents

1. rendre `private` les attributs
2. poser un **verrou** sur les sections critiques : `synchronized`

   section critique = méthode ou bloc d'insctructions

```
public class ...  {
   private Value value;
   public synchronized Value getValue() { return value; }
   public synchronized void setValue(Value value) { this.value = value; }
   ...
}
```

Verrous

- Chaque objet possède un unique verrou (*monitor*)
- Lors de l'invocation d'une méthode "`synchronized`", l'objet est verrouillé : le verrou est acquis par le thread qui l'utilise.
- Aucune méthode "`synchronized`" ne peut être invoquée sur un objet verrouillé : tout autre thread invoquant une telle méthode sera bloqué
- Le verrou est libéré à la sortie de la méthode "`synchronized`" et les threads bloqués peuvent être libérés
- Les invocations de méthodes non "`synchronized`" ne sont pas affectées

```
public class SyncMultipleExclusiveChoice {
   private int currentChoice = 0;
   private boolean choice1 = true;
   private boolean choice2 = false;
   public synchronized void setChoice(boolean choice1, boolean choice2) {
      this.choice1 = choice1;
      this.choice2 = choice2;
   }
   public synchronized boolean validState() {
      return choice1 ^ choice2;
   }
   public void flush() {
      System.out.println(choice1+" "+choice2);
   }
}
```

`Synchronized` (2)

Possibilité de synchroniser sur un bloc d'instructions, en précisant le verrou utilisé

```
Object monitor = ...  ;
synchronized(monitor) {
   // bloc d'instructions
   // exécuté par un seul thread à la fois :
   // celui qui possède le verrou de l'objet monitor
}
```

- ne bloque l'accès qu'aux autres blocs ou méthodes synchronisés sur le **même** objet `monitor`
  (`monitor=this` dans le cas des méthodes synchronisées)
- nécessaire lorsque l'on ne veut pas synchroniser une méthode qui fait des accès à protéger (ex :`run()`)

La synchronisation a un coût !

- ▷ invocation de méthodes synchronisées environ 4 à 6 fois plus lentes que des méthodes non synchronisées
- ▷ occurrences de bloquages inutiles

Synchronisation et collections :

`Collections.synchronizedList(List)`, `Collections.synchronizedSet(Set)`

Il semblerait que cela soit moins efficace

```
import java.util.*;

public class UtilDemo1 {
    public static void main(String args[]) {
        Object obj = new Object();
        // create a list
        List list = new ArrayList();
        // put a wrapper on top of it
        List synclist = Collections.synchronizedList(list);
        // add some objects to the list
        long start = System.currentTimeMillis();
        for (int i = 1; i <= 1000000; i++) {
            synclist.add(obj);
        }
        long elapsed = System.currentTimeMillis() - start;
        System.out.println(elapsed);
    }
}
```

```
import java.util.*;

public class UtilDemo2 {
    public static void main(String args[]) {
        Object obj = new Object();
        // create a list
        List list = new ArrayList();
        // add some objects to it
        long start = System.currentTimeMillis();
        synchronized (list) {
            for (int i = 1; i <= 1000000; i++) {
                list.add(obj);
            }
        }
        long elapsed = System.currentTimeMillis() - start;
        System.out.println(elapsed);
    }
}
```

25% plus rapide que l'exemple d'avant

Deadlock

Si deux threads accèdent de manière concurrente à une ressource synchronisée un blocage (deadlock) peut se produire.

```
package essais.thread;
import java.util.*;
public class Consumer extends Thread {
   private ArrayList resource;
   public Consumer(ArrayList resource) {
      this.resource = resource;
   }
   public Integer consume() {
      synchronized(resource) {
         return (Integer) resource.remove(0);
      }
   }
   public void run() {
      while (true) {
         if (!resource.isEmpty()) {
            System.out.println("consume "+consume());
         }
      }
   }
}
```

```
public class Producer extends Thread {
   private ArrayList resource;
   public Producer(ArrayList resource) { this.resource = resource; }
   public void produce(Integer i) {
      System.out.println("add "+i);
      synchronized(resource) {
         while (!resource.isEmpty()) { }
            resource.add(i);
         }
      }
   public void run() {
      while (true) {
         produce(new Integer((int) (Math.random()*100)));
      }
   }
}
public class DeadLock {
   public static void main(String[] args) {
      ArrayList resource = new ArrayList();
      Producer p = new Producer(resource);
      Consumer c = new Consumer(resource);
      p.start(); c.start();
   }
} // DeadLock
```

```
+-----------------------------
| >java essais.thread.DeadLock
| add 25
| add 77
| [bloqué]
```

Aucun mécanisme ne prévient ce phénomène

... à faire soi-même

- `Thead.sleep(...)` ne libère pas le verrou
- `Thread.yield()` ne libère pas le verrou

`wait()` et `notify()`/`notifyAll()`

Consumer...

```
public Integer consume() {
    Integer i;
    synchronized(resource) {
        i = (Integer) resource.remove(0);
        resource.notify();
    }
    return i;
}
```

Producer...

```
public void produce(Integer i) {
    System.out.println("add "+i);
    synchronized(resource) {
        while (!resource.isEmpty()) {
            try {
                resource.wait();
            }
            catch(InterruptedException e) { e.printStackTrace(); }
        }
    resource.add(i);
    }
}
```

`wait()`, `notify()`/`notifyAll()`

- ▷ méthodes de `Object`
- ▷ doivent être invoquées dans des “zones” synchronisées sur le verrou lui-même
  ↪ sinon : `IllegalMonitorStateException`

wait le thread courant **possède** le verrou sur l’objet, ce thread est mis en attente.
↪ il libère le verrou sur cet objet

notify réveille un seul des threads mis en attente sur le verrou de l’objet
↪ le verrou n’est pas pour autant libéré, il faut que le thread courant sorte de la zone synchronisée

notifyAll tous les threads en attente sont réveillés