Développement Logiciel
L2-S4

Programmation concurrentielle
Thread, multi-thread, synchronisation

anastasia.bezerianos@lri.fr

Les transparents qui suivent sont inspirés du cours de Basé sur :

- le cours du Nicolas Bredeche (Univ. Paris-Sud)

- le cours d'Alexandre Allauzen (Univ. Paris-Sud)

- et quelques transp. du Remi Forax

Point de vue du **système**: Processeur et temps machine

- Du point de vue du système
  - 1+ processeur(s)
  - 1+ coeur(s) par processeur
  - le “temps machine” est partagé entre les programmes

cf. http://download.oracle.com/javase/tutorial/essential/concurrency/procthread.html

Point de vue du **programmeur**: Processus et Thread

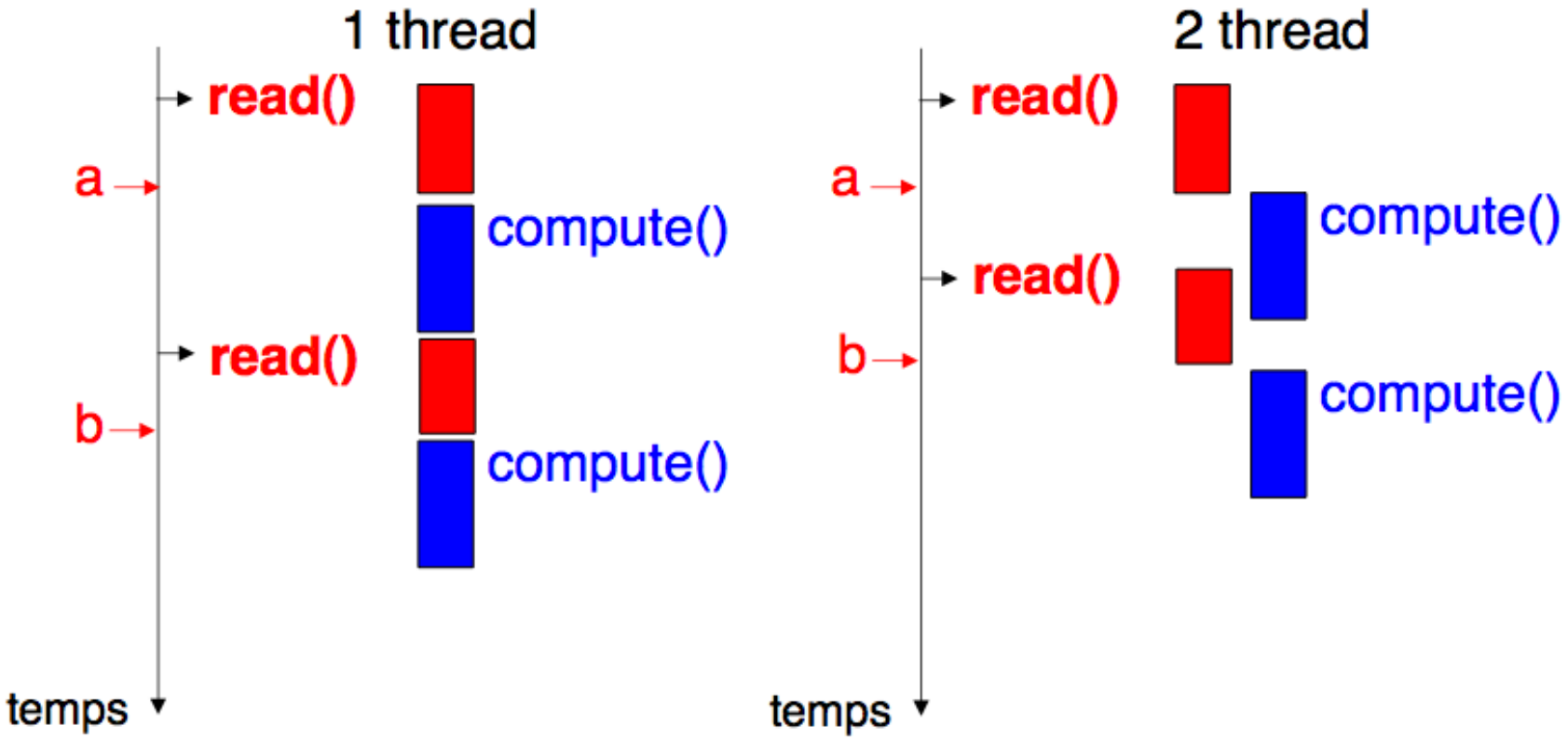
- Deux unités d'exécution:
  - **Processus** :
    - "application", "programme"
    - environnement d'exécution auto-contenu (mémoire, etc.)
  - **Thread**(s) :
    - "processus léger"
    - rattachée(s) à un processus
    - partage les mêmes ressources (mémoire, accès fichiers, etc.)
    - différents branches d'exécution

Point de vue du **programmeur**: Processus et Thread

- Deux unités d'exécution:
  - **Processus**
  - **Thread**(s)

- Programmation concurrentielle:
  - lancer plusieurs threads en parallèle
  - utile si plusieurs tâches concurrentes ou indépendantes
  - bonne idée si contexte multi-processeurs et/ou multi-cores

Exemple : gagner du temps



source: RF

La solution : les threads (1)

- Faire en sorte que la *montre* « tourne » indépendamment de l'application : faire du multi-tâche.
- Pour résumer :
  - Un ordinateur est une machine à exécuter pas à pas un programme.
  - Un programme est une suite d'instructions, son exécution donne lieux à un processus
  - A un moment donné l'ordinateur exécute une et une seule instruction
  - Lorsqu'il y a plusieurs processus : l'ordenanceur répartit le temps de CPU pour chacun.
  - un processus, c'est un PID, un espace mémoire indépendant, un espace d'adressage, un compteur ordinal, ... Gérer par l'OS.

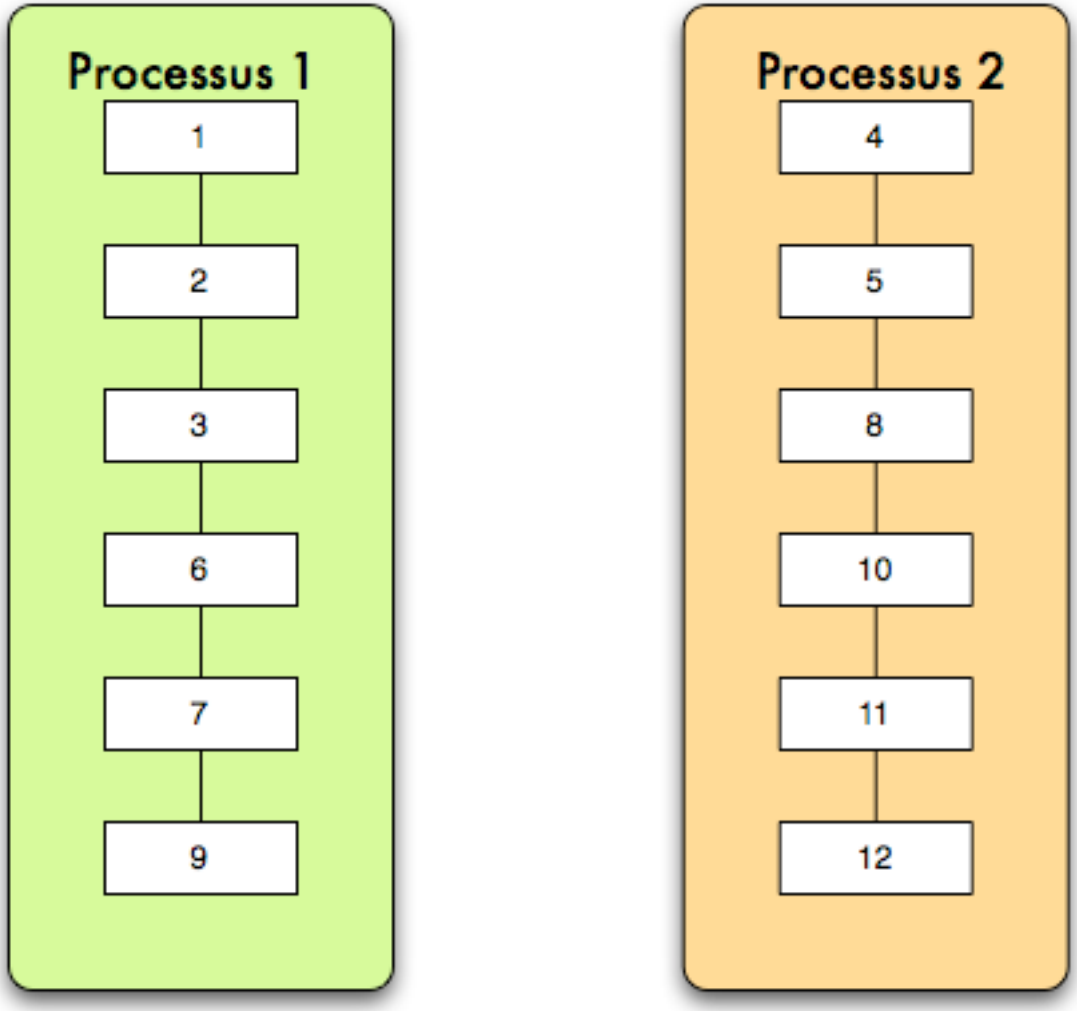
process identifier

La solution : les threads (2)

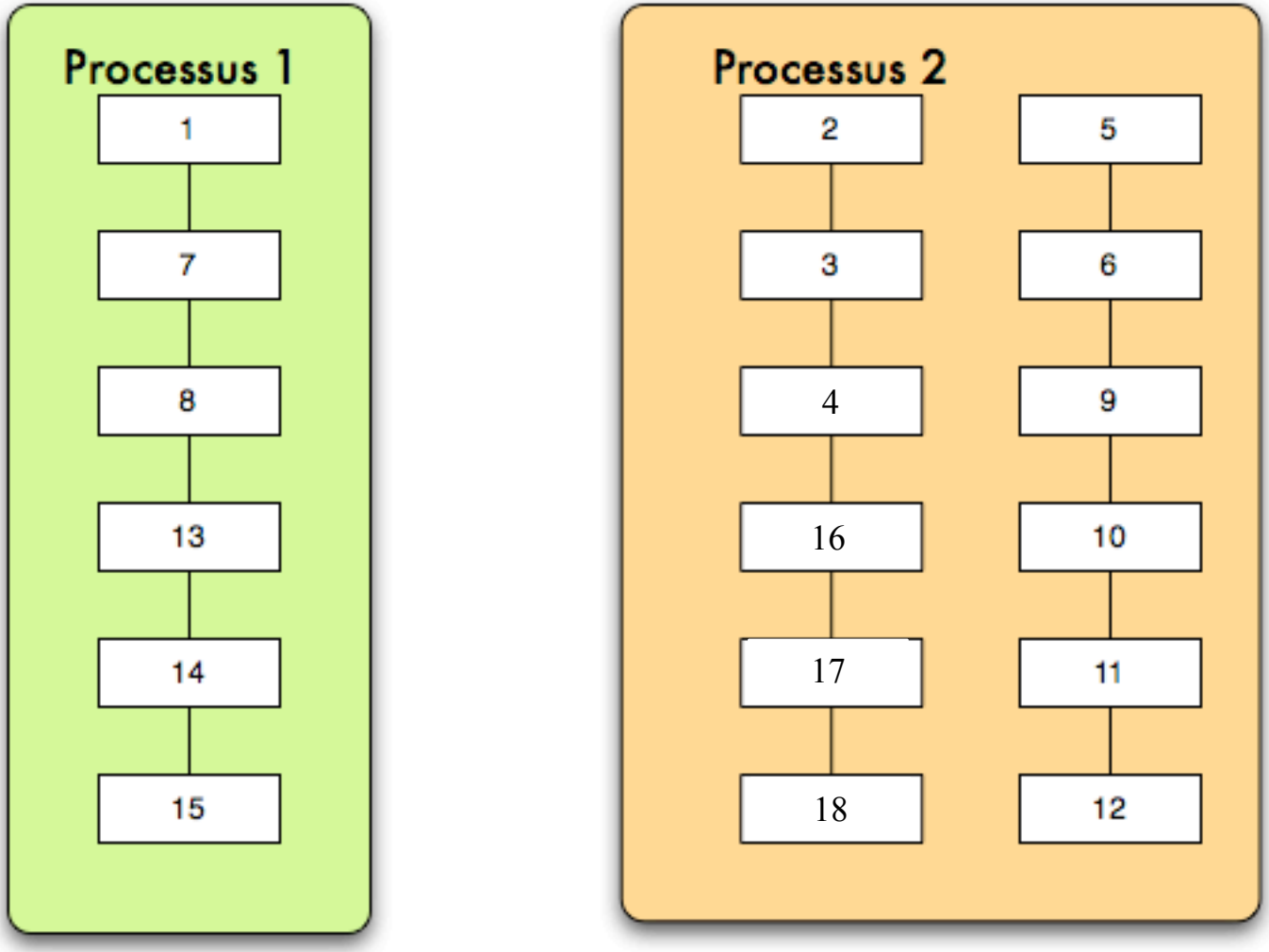
- Faire en sorte que la *montre* « tourne » indépendamment de l'application : faire du multi-tâche.

- En java, il existe des processus « légers » créer par la JVM : les **Threads**.
- Un thread est un « fil » d'exécution, un « sous-processus »
- Un programme java multi-thread :
  - au sein du même processus (*java*)
  - plusieurs fils d'exécution s'exécutent « en parallèle ».
  - L'ordenanceur répartit le temps de CPU entre les processus et les threads.

Exemple : ordonnancement de 2 processus



ordonnancement de processus et threads



Les Threads en java

Les threads en Java

- Il existe toujours un premier thread qui commence à exécuter la fonction *main*()

- Dans les interfaces graphiques
  - Il existe un autre thread qui attend les actions de l'utilisateur et déclenche les écouteurs ("listeners")
  - Il existe un autre thread qui redessine les composants graphiques qui ont besoin de l'être
  - **=> cf. prochains cours**
- On peut créer ses propres threads

La classe **Thread** et l'interface **runnable** (1)

- La création d'un thread passe par la création d'un objet de la classe java.lang.Thread.
- Un objet de type Thread représente la télécommande d'un thread réel
- Il sert à le manipuler (contrôle, priorité, synchronisation)
- Il faut indiquer au thread quelle méthode exécuter

  (met pointeur sur méthode ? : ⇒ faire un ***run*** de quoi ?)

  ... (suite)

La classe **Thread** et l'interface **runnable** (2)

... (suite)

- Créons un objet qui soit « runnable », il sera le cible du thread.
- Un objet runnable implémente l'interface *Runnable*.

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

- La méthode *run* est la liste des tâches d'un thread, son « main ».

Tout thread: but est l'exécution d'une méthode **run**

Création et démarrage - 1

- Une fois créer, un thread ne fait rien tant qu'il n'a pas commencé : avec **start**().
- **start**() lance l' exécution de **run**().
- Une fois lancer, un thread va jusqu'au bout de **run**(), du moins tant qu'on ne le stop pas : **stop**().

```
class Animation implements Runnable {
    ...
    void run() {
        while ( true ) {
            // Draw Frames
            ...
            repaint();
        }
    }
}
```

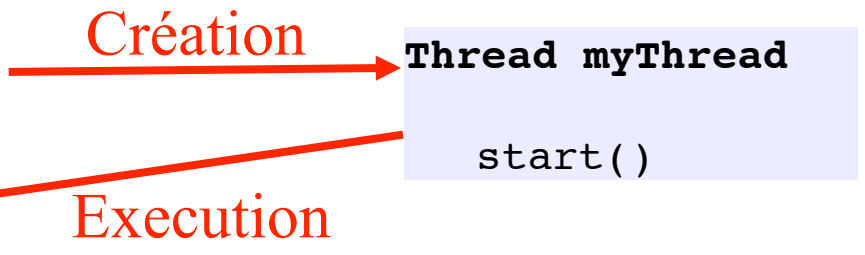
Une première manière, pas très objet.

```
Animation happy = new Animation("Mr.Happy");
Thread myThread = new Thread( happy );
// Jusqu'ici rien n'est lancé !
myThread.start();
```

Création et démarrage - 2

- Il est préférable, qu'un objet contrôle son propre thread
- L'objet est alors autonome
- C'est la bonne solution, la plus fréquemment utilisée

```
class Animation implements Runnable {

   Thread myThread;

   // par exemple dans le contructeur
   Animation (String name) {
     myThread = new Thread(this);
     myThread.start();
   }
   ...
   public void run() {...}
}
```



Création et démarrage - 3

- On peut rencontrer cependant la solution suivante :
  - hériter de la classe *Thread* qui implémente *Runnable.*
  - redéfinir la méthode **run**().

```
class Thread implements Runnable
    .....
}
```

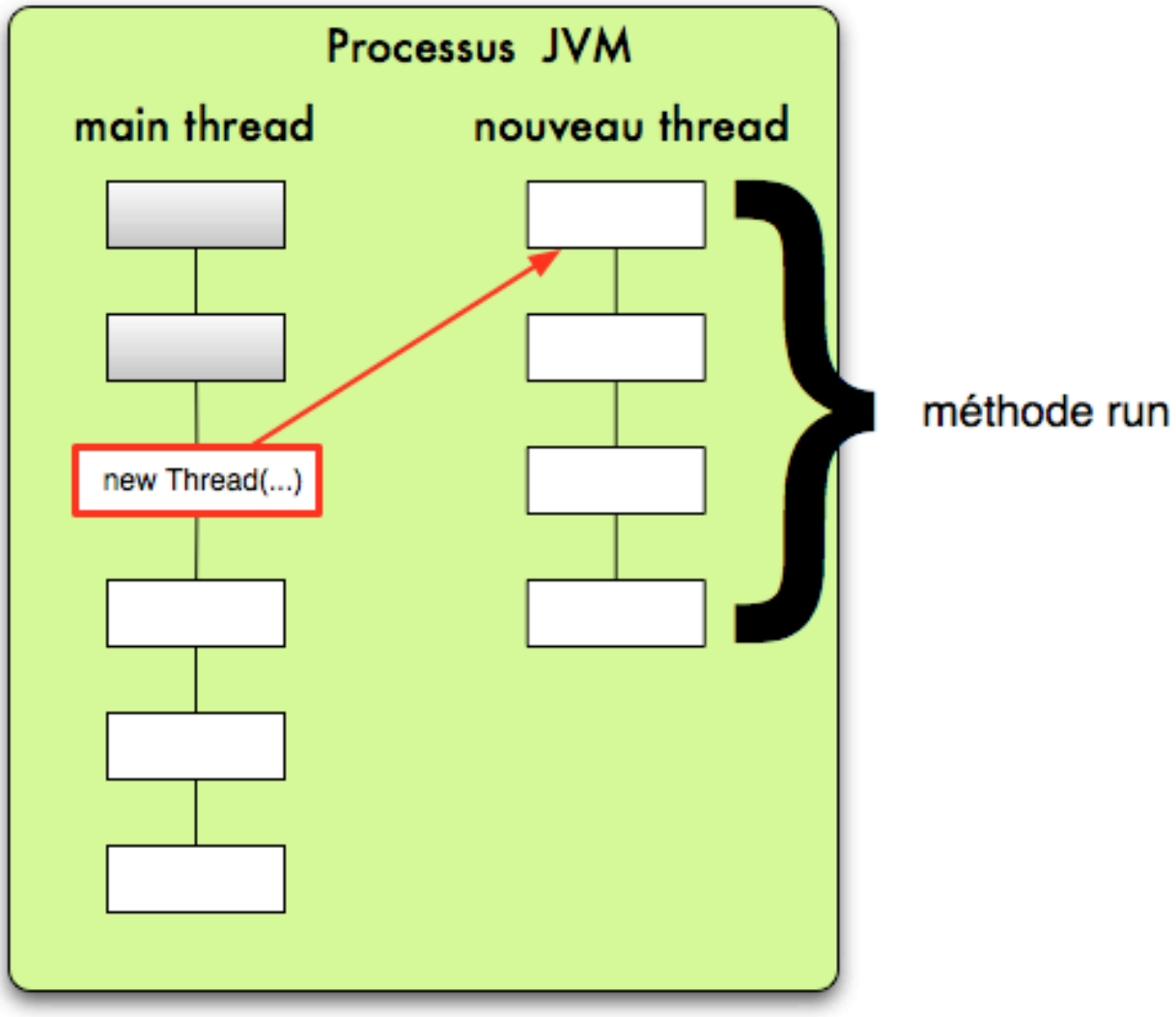
ou bien

```
class Animation extends Thread{
   ...
   public void run() {...}
}
```

```
class Animation extends Thread{
   // ...
   Animation (String name) {
        start();
    }
   public void run() {...}
}
```

```
Animation fun = new Animation("fun");
fun.start();
```

Thread : illustration



Une vision d'un thread

```
public class Thread implements Runnable{
   private Runnable target;
   ...

   public Thread(){
     ...
     target = this;
   }

   public void start(){
     target.run();
   }
```

Reprenons : utilisation de la classe *Thread*

- *« The Thread class implements a standard thread that, by default, does nothing. »*
- Il faut redéfinir la méthode *public void run()* qui décrit la tâche de l'objet de type *Thread*.
- Un thread démarre avec l'appel de la méthode *start() qui lance run().*
- Pour réutiliser, une première possibilité est d'hériter de *Thread* (mais notre application peut hériter déjà)
- La seconde possibilité est d' implémenter l'interface Runnable et construire un thread qui a pour cible un objet Runnable.
  - Il existe l'interface Runnable qui impose la méthode run().
  - Un objet Runnable contient ce que le thread doit exécuter.

Hériter de *Thread ou non.*

- Hériter de ***Thread*** semble simple, ... mais il est important de concevoir vos classes selon une structure objet cohérente :
  - la méthode *run()* est souvent relative à la fonctionnalité d'une classe particulière (par exemple la montre, l'animation, ...).
  - *run()* devrait être implémentée dans cette classe, ... plutôt que dans une classe représentant un thread.

- Implémenter l'interface ***Runnable*** permet :
  - de faire un design objet cohérent.
  - d'hériter d'une autre classe que *Thread* (*JFrame*, *JLabel*);
  - permettre à *run()* d'accéder aux membres privés de la classe.

Contrôle des threads

- La vie d'une thread démarre par l'appel de *start*() qui lance *run*().
- *stop*() détruit le thread en cours.
- *start*() et *stop*() ne peuvent être appelé qu'une fois !
  ... au contraire de *suspend*() et *resume*().
- Pour la plupart des utilisations *start*() et *stop*() suffisent, c'est lorsque quand l'initialisation du thread est importante, qu'il faut utiliser s*uspend*() et *resume*() : connexion réseau, ressource chargée importante, ...
- Le sommeil *sleep*(milisecondes):

```
try {
        Thread.sleep(1000);
} catch (InterruptedException e) {
}
```

Exemple pratique: la classe Montre

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Montre implements Runnable{
    ...
    public void run(){
        while (true) {
            try {
                Thread.sleep(1000);
            } catch(InterruptedException e) {
                System.err.println("Interrupted");
            }
            this.setText(new java.util.Date().toString());
        }
    }
}
```

EXERCICE : la course des garçons de café

Jeu de la «Course de Garçons de Café» du Musée des Arts Forains

le signal est codé dans le main. le gagnant y est testé. la méthode run() de chaque garcon de café contient le code:

```
while ( true )
{
   if (start == true)
   {
     pos += Math.random();
     ...etc.
```

- Il y a 4 garçons de café. Chaque garçon de café
  - part du km 0, et doit atteindre le km 10
  - parcours entre 0 et 1 km à chaque appel de thread (utiliser Math.random())
  - le premier qui arrive au km 10 a gagné
  - tous attendent le *même signal* pour partir

un thread *daemon*

- Propose un service général en tâche de fond.
- Un programme Java s'arrête une fois que tous les threads sont terminés.
  - tous sauf les daemon.
  - i.e. demons en statue *run* n'empeche pas le programme de terminer; les autres threads en statue *run* empeche le programme de terminer.
- Transformer un thread en daemon :
  - `setDaemon(true);`
  - `isDaemon();`
- Pour la montre, mieux qu'un thread, un daemon.

Synchronisation et accès concurrent

Threads en parallèle: exemple

```
public class SimpleThread extends Thread {
  private int countDown = 5;
  private static int threadCount = 0;
  private int threadNumber = ++threadCount;
  public SimpleThread() {
    System.out.println("Making " + threadNumber); }
  public void run(){// la méthode à redéfinir : la tâche du thread
    while(true) {
      System.out.println("Thread "+threadNumber+"("+countDown+")");
      if(--countDown == 0) return;
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new SimpleThread().start();//le start lance le thread (run)
    System.out.println("All Threads Started");
  }
```

du livre de Bruce Eckel

execution de simple*Thread*

```
Making 1
Making 2                    Thread 3(5)
Making 3                    Thread 4(5)
Making 4                    Thread 4(4)
Making 5                    Thread 4(3)
Thread 1(5)                 Thread 4(2)
Thread 1(4)                 Thread 4(1)
Thread 1(3)                 Thread 5(5)
Thread 1(2)                 Thread 5(4)
Thread 2(5)                 Thread 5(3)
Thread 2(4)                 Thread 5(2)
Thread 2(3)                 Thread 5(1)
Thread 2(2)                 Thread 3(4)
Thread 2(1)                 Thread 3(3)
Thread 1(1)                 Thread 3(2)
All Threads Started         Thread 3(1)
```

le résultat sur une autre machine (autre OS).

```
Making 1
Making 2
Making 3
Making 4
Thread 1(5)
Thread 1(4)
Thread 1(3)
Thread 1(2)
Thread 1(1)
Thread 2(5)
Thread 2(4)
Thread 2(3)
Thread 2(2)
Thread 2(1)
Thread 3(5)
Thread 3(4)
Thread 3(3)
Thread 3(2)
Thread 3(1)
Making 5
All Threads Started
Thread 4(5)
Thread 4(4)
Thread 4(3)
Thread 4(2)
Thread 4(1)
Thread 5(5)
Thread 5(4)
Thread 5(3)
Thread 5(2)
Thread 5(1)
```

le résultat sur une autre machine (autre OS).

```
Making 1
Making 2
Making 3
Making 4
Thread 1(5)
Thread 1(
Thread 1(
Thread 1(
Thread 1(1)
Thread 2(5)
Thread 2(4)
Thread 2(3)
Thread 2(2)
Thread 2(1)
Thread 3(5)
Thread 3(4)
Thread 3(3)
Thread 3(2)
Thread 3(1)
Making 5
                    4(3)
Thread 4(2)
Thread 4(1)
Thread 5(5)
Thread 5(4)
Thread 5(3)
Thread 5(2)
Thread 5(1)
```

Comment exécuter les threads en séquence?

Planification et *yield*

- Un thread peut donner de son temps volontairement par l'appel de *yield*().
- Pour faire alterner des threads, c'est la bonne solution.

- S'il s'agit de passer la main : yield().
- La méthode sleep() doit servir que lorsque l'on souhaite imposer un temps donner de sommeil.

`static void`	**`yield`**`()` Causes the currently executing thread object to temporarily pause and allow other threads to execute.

Avec yield()

du livre de Bruce Eckel

```
public class SimpleThread extends Thread {
  private int countDown = 5;
  private static int threadCount = 0;
  private int threadNumber = ++threadCount;
  public SimpleThread() {
    System.out.println("Making " + threadNumber); }
  public void run() {
    while(true) {
      System.out.println("Thread "+threadNumber+"("+countDown+")");
      if(--countDown == 0) return;
      yield();
    }
  }
 ........
}
```

le résultat avec *yield()*

```
Making 1
Making 2
Making 3
Making 4
Making 5
Thread 1(5)
Thread 2(5)
Thread 3(5)
All Threads Started
Thread 4(5)
Thread 1(4)
Thread 2(4)
Thread 3(4)
Thread 5(5)
Thread 4(4)
Thread 1(3)
Thread 2(3)
Thread 3(3)
Thread 5(4)
Thread 4(3)
Thread 1(2)
Thread 2(2)
Thread 3(2)
Thread 5(3)
Thread 4(2)
Thread 1(1)
Thread 2(1)
Thread 3(1)
Thread 5(2)
Thread 4(1)
Thread 5(1)
```

Problème de manipulation

Exemple : Feuille de calcul, tableau dynamique

- Que se passe-t-il si deux threads manipulent un même objet ?

```
public class Counter {
    private int c = 0;
    public void increment() {c++;}
    public void decrement() {c--;}
    public int value() {return c;}
} // c++, c--: récupèrent c, le modifie, et le sauvegardent

// suppose Tread A increment() et Tread B decrement()
// Un résultat possible :
Thread A: Retrieve c.
Thread B: Retrieve c.
Thread A: Increment retrieved value; result is 1.
Thread B: Decrement retrieved value; result is -1.
Thread A: Store result in c; c is now 1.
Thread B: Store result in c; c is now -1.
```

Le résultat de Thread A est perdu, et cette comportement peut changer!

Problème de manipulation

Exemple : Feuille de calcul, tableau dynamique

- Que se passe-t-il si deux threads manipulent un même objet ?

```
public class Counter {
    private int c = 0;
    public void increment() {c++;}
    public void decrement() {c--;}
    public int value() {return c;}
} // c++, c--                                    regardent

// suppose Tr                                    ()
// Un résulta
Thread A: Retrieve c.
Thread B: Retrieve c.
Thread A: Increment retrieved value; result is 1.
Thread B: Decrement retrieved value; result is -1.
Thread A: Store result in c; c is now 1.
Thread B: Store result in c; c is now -1.
```

Comment synchroniser l'accès par les threads?

Le résultat de Thread A est perdu, et cette comportement peut changer!

Problème de manipulation

Exemple : Feuille de calcul, tableau dynamique

```
public class SpreadSheet {

    int cellA1, cellA2, cellA3;

    public int sumRow() {
        return cellA1 + cellA2 + cellA3;
    }

    public void setRow( int a1, int a2, int a3 ) {
        cellA1 = a1;
        cellA2 = a2;
        cellA3 = a3;
    }
...
}
```

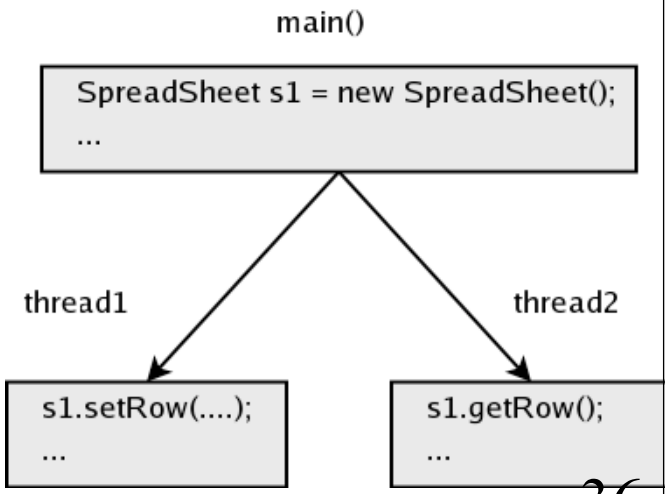
- Que se passe-t-il si deux threads manipulent un même objet de type `SpreadSheet` et appellent au même moment `sumRow()` et `setRow()` ?

Synchronisation

- Prévenir les collisions au niveau de l'objet et de sa mémoire.
- Déclarer une méthode ou un bloc de code *synchonized*
- Pour un objet donné:
  - Un seul verrou (*lock*) est partagé par toutes les méthodes *synchronized*.
  - Donc: un seul Thread à la fois peut appeler une méthode *synchronized*

Principe d'encapsulation + synchronization :

l'accès aux membres privés se fait via des méthodes synchronized (les accesseurs).



Exemple : spreadSheet (suite)

```
public class SpreadSheet {

    int cellA1, cellA2, cellA3;

    public synchronized int sumRow() {
        return cellA1 + cellA2 + cellA3;
    }

    public synchronized void setRow( int a1, int a2, int a3 ) {
        cellA1 = a1;
        cellA2 = a2;
        cellA3 = a3;
    }
...
```

- Toutes les méthodes doivent être déclarée `synchronized`
- Si un verrou est posé, les autres taches attendent la levée du verrou (pour **tous les méthodes de l'objet**).

Limitation des méthodes syncronized

```
private List<Foo> myList = new ArrayList<Foo>();
private Map<String,Bar> myMap = new HashMap<String,Bar>();

public synchronized void Map_put( String s, Bar b ) {
    myMap.put( s,b );
}

public synchronized void List_add( Foo f ) {
    myList.add( f );
```

• Si un verrou est posé, les autres taches attendent la levée du verrou (pour **tous les méthodes de l'objet**).
• Mais si les méthodes essayez d'accéder à différentes parties de l'objet?

Critical section

- Pour partiellement synchroniser une partie d'une méthode : du code « critique »

```
public void run() {
    while (true) {
      synchronized(this) {
        t1.setText(Integer.toString(count1++));
        t2.setText(Integer.toString(count2++));
      }
      try {
        sleep(500);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
```

Synchronisation généralisée

- Synchroniser ce que l'on veut, sur ce que l'on veut.

```
synchronized(syncObject){
    /* Ce code ne peut être exécuter que par
       un seul thread à la fois */
    ...
}
```

- Ainsi les 2 morceaux de codes suivants sont équivalents (car on a un verrou pour l'accès à this).

```
synchronized void myMethod () {
    ...
}
```

```
void myMethod () {
    synchronized (this) {
        ...
    }
}
```

Synchronisation généralisée

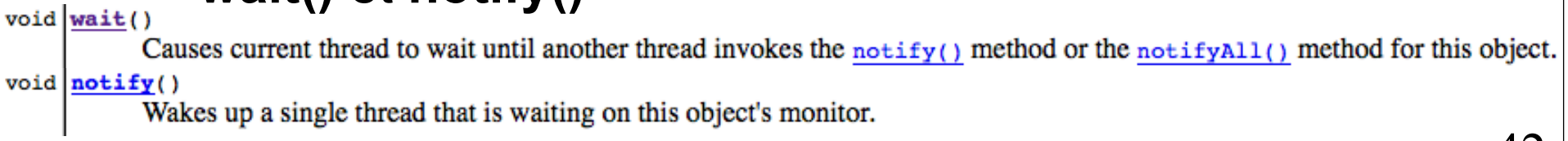
```
private List<Foo> myList = new ArrayList<Foo>();
private Map<String,Bar) myMap = new HashMap<String,Bar>();

public void Map_put( String s, Bar b ) {
  synchronized( myMap ) {
    myMap.put( s,b );
    ...
  }
}
public void List_add( Foo f ) {
  synchronized( myList ) {
    myList.add( f );
    ...
  }
}
```

- Dans le même objet nous pouvons avoir plusieurs verrous
- Nous appelons les objets qui fonctionnent comme verrous "moniteurs" (monitors)

Interblocage

- Que se passe t'il quand 2 threads s'attendent mutuellement?
  - Dans une méthode synchronized
    - Le verrou est bloqué à l'entrée
    - ... puis débloqué à la sortie
    - Si au milieu de la méthode un thread se rend compte qu'il ne peux pas remplir ses objectifs vue un autre Thread bloqué, doit il avancer ?

      » ...INTERBLOCAGE
- Solution?
  - Mieux vaut passer la main si on ne peut plus avancer.
  - **wait()** et **notify()**



void **wait**()
Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

void **notify**()
Wakes up a single thread that is waiting on this object's monitor.

Exemple pratique: Interblocage et résolution

- Exemple classique du producteur/consommateur (producer/consumer)
  - Le producteur fonctionne en flux tendu avec un stock très limité.
  - Le consommateur consomme le produit à son rythme (il est parfois un peu lent).
  - Le producteur doit attendre le consommateur pour produire.

Producteur / Consommateur

- Le producteur :
  - contient un Vector (*collection*) des messages
  - le thread Producteur ajoute un message au vecteur par seconde
  - Mais le vecteur a une taille maximale
    - Dans ce cas la méthode *putMessage* «tourne en rond».
- Le consommateur :
  - consomme un message toutes les 2 secondes avec la méthode *getMessage* du producteur.

- Synchronisation : les deux méthodes *getMessage* et *putMessage* accède au même vecteur ! Il faut donc les synchroniser.

Le producteur (1)

```
import  java.util.Vector;

public class Producer extends Thread {
    static final int MAXQUEUE = 5;
    private Vector messages = new Vector();

    public void run() {
        try {
            while ( true ) {
                putMessage();
                sleep( 1000 );
            }
        } catch( InterruptedException e ) { }
    }
```

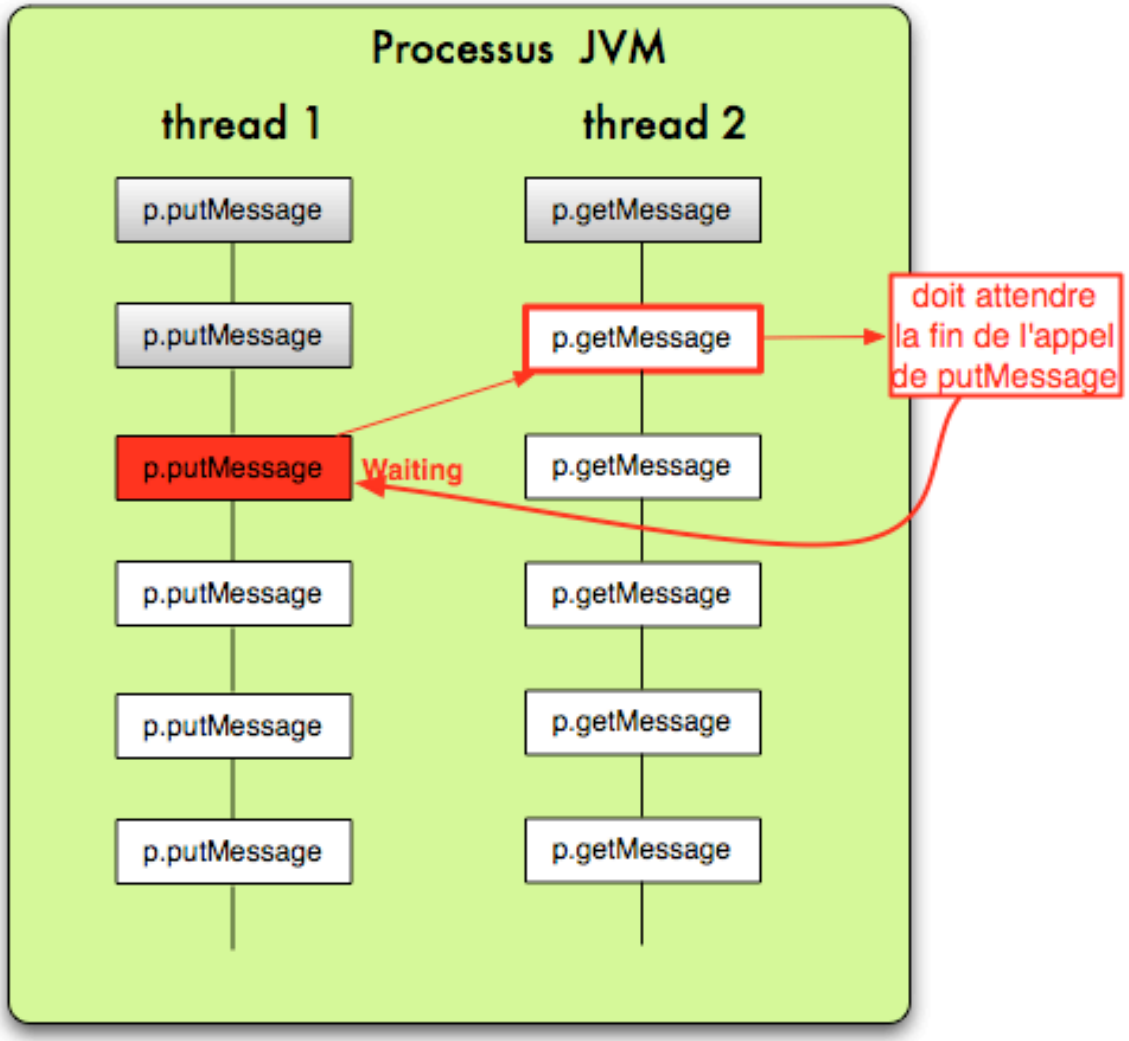
Le producteur (2)

```
...
   private synchronized void putMessage()
                           throws InterruptedException {
       while (messages.size() == MAXQUEUE){
          System.out.println("Waiting”);
       }
       messages.addElement(new java.util.Date().toString());
   }
   // Called by Consumer
   public synchronized String getMessage()
                          throws InterruptedException {
         while ( messages.size() == 0 )
          try {
             sleep( 1000 );
          } catch(InterruptedException e){}
         String message = (String) messages.firstElement();
         messages.removeElement(message);
         return message;
     }
}
```

Le consommateur

```
import  java.util.Vector;
public class Consumer extends Thread {
    Producer producer; // quel est son producteur.

    Consumer(Producer p) { producer = p; }
    public void run() {
        try {
            while ( true ) {
                String message = producer.getMessage();
                System.out.println("Got message: " + message);
                sleep( 2000 ); // il est un peu lent.
            }
        } catch( InterruptedException e ) { }
    }
    public static void main(String args[]) {
        Producer producer = new Producer();
        producer.start();
        new Consumer( producer ).start();
    }
}
```

Interblocage



Le producteur v.2 (1)

```
import  java.util.Vector;
public class Producer extends Thread {
    static final int MAXQUEUE = 5;
    private Vector messages = new Vector();

    public void run() {
        try {
            while ( true ) {
                putMessage();
                sleep( 1000 );
            }
        }  catch( InterruptedException e { }
    }

     // idem
```

Le producteur v.2 (2)

```
...
    private synchronized void putMessage()
                throws InterruptedException {
        while (messages.size() == MAXQUEUE){
             wait(); // Attendons (debloquer)
             System.out.println("Waiting”);
        }

        // Reprenons
        messages.addElement(new java.util.Date().toString());
        notify(); // debloquer et notifions prochaine thread
    }
```

Le producteur v.2 (3)

```
    // Called by Consumer
    public synchronized String getMessage()
        throws InterruptedException {
        while ( messages.size() == 0 )
            wait(); // Attendons (debloquer)

        // Reprenons
        String message = (String) messages.firstElement();
        messages.removeElement(message);
       notify();  // Notifions prochaine
        return message;
    }
}
```

EXERCICE

- Ecrivez
  - un thread qui écrit entre 3 et 14 caractères au hasard dans un String, avec des pauses allant de 0 à 4 secondes. Uniquement lorsque le String est vide.
  - un thread qui affiche le contenu de ce String à l'écran, uniquement lorsqu'il est plein, et en le vidant ensuite.

wait en détail

- *wait* est héritée de la classe *Object*.
- Cette instruction s'utilise dans un code synchronisé.
- Le thread en cours d'exécution attend. La méthode doit être invoquée sur l'instance verrouillée ("code critique" ou méthode synchronisé). Typiquement, on pourra écrire :

```
synchronized(unObjet){
   ...
   while (condition)
       unObjet.wait();
   ... }
```

- Le thread en cours d'exécution est bloqué par *unObjet*.
- L'instruction unObjet.wait() signifie que unObjet bloque le thread en cours d'exécution (ce n'est pas unObjet qui attend).
- Pendant que la thread attend, le verrou sur unObjet est relâché.

notify en détail

- Pour pouvoir débloquer un thread qui attend, bloqué par unObjet

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

- Si plusieurs threads sont dans l'attente imposée par *unObjet*, l'instruction *notify*() débloque celle qui attend depuis le plus longtemps.
- Sinon *notifyAll()*.
- Signalons qu'il existe aussi *wait(long temps)* qui termine l'attente après temps en millisecondes, si cela n'est pas fait auparavant.

Planification des threads

- Lorsqu'un thread est lancé, il garde la main jusqu'à ce qu'il atteigne un des états suivants :
  - Sleeps : appel de Thread.sleep() ou wait().
  - Waits for lock : une méthode synchronized est appellée, mais le thread n'a pas le verrou.
  - Blocks on I/O : lecture de fichier, attente reseau.
  - Explicit yield control : appel de yield().
  - Terminates : la méthode run() du thread est terminé ou appel de stop().
- Qui prend la main ensuite ?

Gestion des priorités

- Java propose quelques garanties sur comment les threads sont planifiées (scheduled).
- Chaque thread possède une priorité.
  - Si 2 Threads se battent pour obtenir la main …
  - Si un Thread de plus haute priorité veut la main il l'obtient
  - Si 2 Threads de même priorité se battent : Time slicing
- Un thread hérite de la priorité de son créateur.

- La priorité peut être modifiée :

```
int myPriority = Thread.MIN_PRIORITY;
thread1.setPriority(myPriority+1);
```

- La classe Thread posède les constantes Thread.MIN_PRIORITY (resp. MAX et NORM).

Je veux terminer mon travail !

- Adhérer à un thread
  - Un thread peut appeler *join ()* sur un autre thread et l'attendre (à terminer) avant de poursuivre.
- Si un thread appelle *t.join ()* sur un autre thread *t*,
  - le thread appelant est suspendu jusqu'à la fin du cible thread *t* (quand *t.isAlive ()* est faux).

- Nous pouvons aussi appeler *join ()* avec un argument timeout (en milliseconds), de sorte que si le thread cible ne se termine pas en cette période de temps, notre thread continue son exécution.

Timer : Simple et Utile

- Pragmatiquement
  - Un Thread pour repousser une action dans le temps
    - Dans 3 secondes, si l'utilisateur n'a pas bougé sa souris, afficher un popup disant « Ce bouton sert à quitter le document »
  - Un Thread pour répéter une action régulièrement
    - Tous les 0.5 secondes, redessine la barre de progression.

- Pour ces cas simple, pas besoin de faire des Threads compliqués : Utiliser un Timer !

fin du cours sur la programmation concurrentielle