Année 2008-2009

esil

Les threads Java

Nicolas Baudru
mél : nicolas.baudru@esil.univmed.fr
page web : nicolas.baudru.perso.esil.univmed.fr

Revenons quelques instants à nos animaux.

```
public class Chien {
  public void aboyer()  {
    // ici  se trouve le comportement normal d'un chien
  }
}

public class TestChien {
  public static void main ( String[] args ) {
    Chien c1= new Chien();
    Chien c2= new Chien();
    c1.aboyer();
    c2.aboyer();
  }
}
```

Les deux chiens aboient-ils en même temps ?

En java, il est possible de simuler l'exécution de plusieurs programmes « en même temps ». C'est le multithreading. L'exécution de chaque programme est alors appelé un thread.

Le multithreading est intégré au langage Java. La création de threads est très simple :

```
Thread t = new Thread(); // créer un nouveau thread
t.start();
```

En créant un nouvel objet Thread, vous créez un fil d'exécution séparé qui possède sa propre pile.

A votre avis, que fait ce thread ?

Tâche d'un thread

En Java le multithreading signifie qu'il faut

- créer un thread,
- écrire la tâche exécutée par le thread,
- lier la tâche au thread.

Pour réaliser cela, il faut étudier et comprendre la classe Thread.

Thread
void join() void start() static void sleep()

Que signifie avoir plusieurs piles d'appels

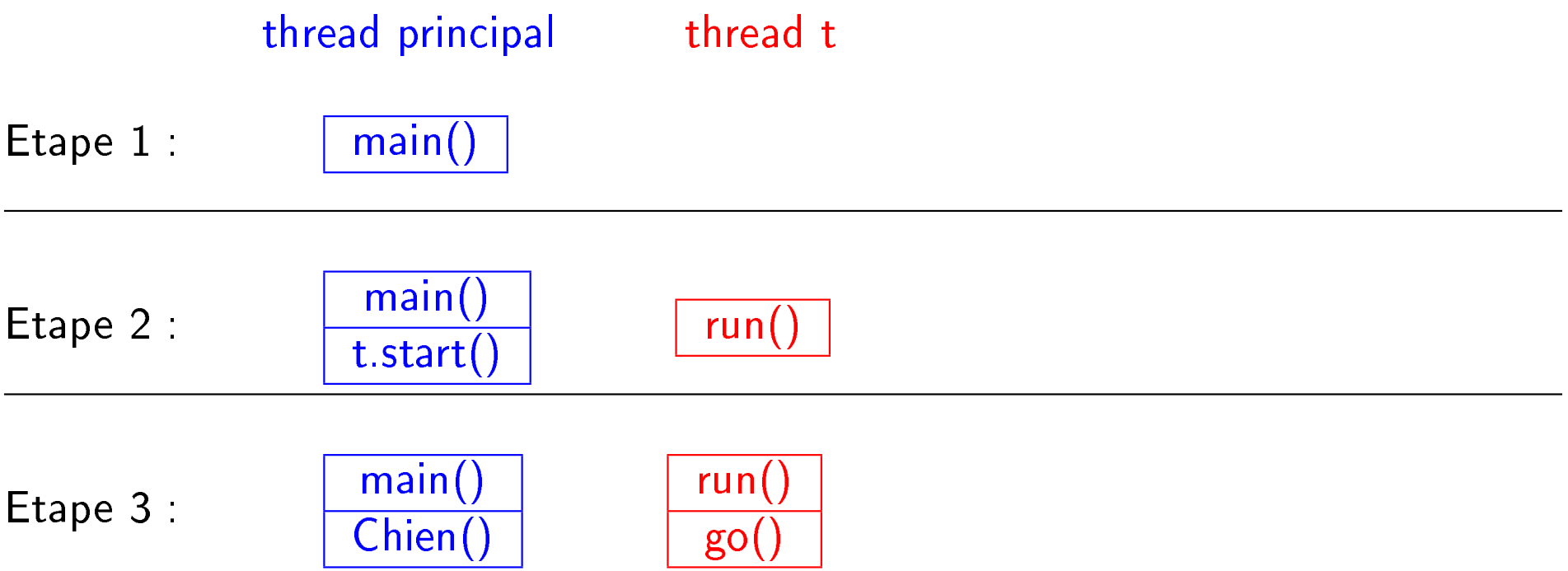
esil

Avec plusieurs piles d'appels, on a l'impression que plusieurs choses se passent en même temps. Dans un système multiprocesseurs cela serait réellement possible, mais pas sur un système monoprocesseur. Il faut donc « faire semblant », ce qui est rendu possible par les threads.

A votre avis comment Java peut donner l'impression d'exécuter plusieurs piles d'appels ?

Exemple

```
public static void main (String [ ] args ) {  // étape 1
   Runnable r = new MaTache();
   Thread t = new Thread(r);
   t.start(); // étape 2 : une nouvelle pile est créée
   Chat c = new Chat();
}
```



Comment lancer un nouveau thread

1 Créer un objet Runnable (la tâche du thread) :

```
Runnable threadTache = new MonRunnable();
```

En fait, Runnable est une interface. Il faut donc écrire une classe implémentant cette interface et définissant la tâche que le thread doit exécuter dans une nouvelle pile. Ici, cette classe est appelée MonRunnable.

2 Créer un objet Thread et lui passer une tâche :

```
Thread monThread = new Thread(threadTache);
```

On passe l'objet Runnable au constructeur de Thread. Le thread sait ainsi quelle méthode placer au sommet de la pile : c'est la méthode run() de l'objet Runnable.

3 Lancer le thread :

```
monThread.start();
```

C'est à ce moment là qu'un nouveau fil d'exécution est créé. Une nouvelle pile est créée avec la méthode run() en son sommet.

L'interface Runnable ne définit qu'une seule méthode : la méthode void run().

Lorsque vous voulez définir une tâche, il vous suffit d'écrire une classe qui implémente Runnable. Cette classe contient au moins une méthode.

Laquelle et pourquoi ?

Ainsi, lorsque vous fournissez à un constructeur de Thread un objet Runnable, vous lui donnez un moyen d'obtenir une méthode run(). Autrement dit, vous lui donnez du travail à faire.

Exemple

```
public class MonRunnable implements Runnable {
  public void run() {   go();   }
  public void go() {   // faire des choses }
}

class TestThread {
  public static void main ( String[]  args) {
    Runnable tache = new MonRunnable();
    Thread monThread = new Thread(tache);
    monThread.start();

    // d'autres trucs à faire ...
  }
}
```

Etats des threads

1 le thread est nouveau. L'objet est créé mais il n'y a pas encore de fil d'exécution.

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

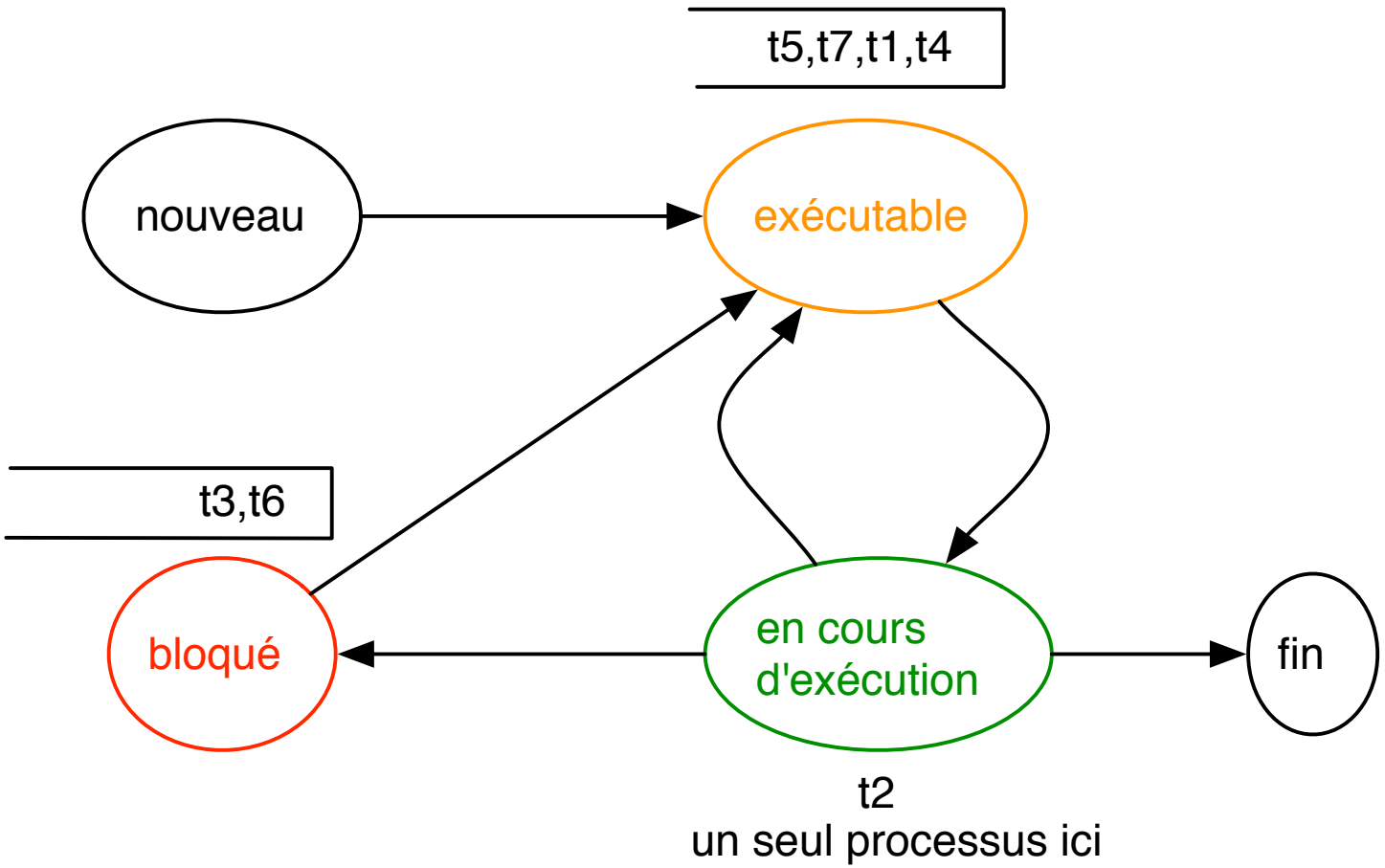
2 le thread est exécutable. Quand vous lancez le thread, une pile est créée. Le thread attend alors que la JVM le choisisse afin de s'exécuter.

```
t.start;
```

3 le thread est en cours d'exécution. C'est LE thread qui s'exécute, le seul. C'est l'ordonnanceur de la JVM qui décide quel thread va s'exécuter à un moment donné. Quand un thread s'exécute les autres sont en attente. A tout moment l'ordonnanceur peut arrêter l'exécution du thread (il retourne alors dans l'état exécutable) et en choisir un autre.

4 le thread est bloqué. Le thread veut obtenir une ressource (une méthode d'un objet « verrouillé », le clavier, ...) non disponible pour le moment.

```
ex : t.sleep(200);
```


t5,t7,t1,t4
nouveau
exécutable
t3,t6
bloqué
en cours
d'exécution
fin
t2
un seul processus ici

Quel est le résultat du programme suivant ?

```
public class MonRunnable implements Runnable {
  public void run() {   go();   }
  public void go() {
    System.out.println("␣monThread");
  }
}

class TestThread {
  public static void main ( String[]  args) {
    Runnable tache = new MonRunnable();
    Thread monThread = new Thread(tache);
    monThread.start();
    System.out.println ("retour␣à␣main()");
  }
}
```

Peut-on ressusciter un thread mort (i.e. qui a terminé son exécution) ?

Endormir un thread

La méthode statique sleep(int d) de la classe Thread force le thread en cours d'exécution à passer dans l'état bloqué pendant d ms. L'ordonnanceur choisira alors un autre thread dans la liste des threads exécutables.

```
public class MonRunnable implements Runnable {
   public void run() {   go();   }
   public void go() {
     try {
       Thread.sleep(2000);
     } catch (InterruptedException ex) {
       ex.printStackTrace ();
     }
     // faire d'autres trucs
   }
 }
```

Au bout de combien de temps un thread bloqué par un appel à sleep va-t-il retrouver son état « en cours d'exécution » ?

Encore un exemple

```
public class MaTache implements Runnable {
  public void run () {
    for (int i = 0; i < 25; i++) {
      String nom = Thread.currentThread().getName();
      System.out.println(nom +
                "est␣en␣cours␣d'exécution");
  }

  public static void main ( String[] args ) {
    Runnable tache = new MaTache();
    Thread a = new Thread(tache);
    a.setName("Caroline");
    Thread b = new Thread(tache);
    b.setName("Cedric");
    a.start();
    b.start();
  }
}
```

Quel est le résultat ?

Une situation où deux processus ou plus lisent ou écrivent des données partagées et où le résultat final dépend de quel élément s'exécute à un moment donnée est appelée le problème de l'accès concurrent.

Exemple : voir complément de cours

Il est très difficile de déboguer des programmes contenant des problèmes de concurrence, car les résultats des tests sont souvent corrects à première vue, mais de temps en temps il se produit des situations curieuses et inexpliquées.

Pour éviter les problèmes de ce type de problème, il faut trouver une solution pour interdire que plusieurs threads lisent et écrivent des données partagées simultanément.

L'exclusion mutuelle est une méthode qui permet de s'assurer que si un thread utilise une variable ou un fichier partagés, les autres processus seront excluent de la même activité. Les choix de la mise en oeuvre de l'exclusion mutuelle est une question de conception majeure pour tout système d'exploitation. Leur étude sort de la portée de ce cours.

Dans un programme, la partie à partir de laquelle on accède à une ressource partagée se nomme section critique. Pour éviter les problèmes d'accès concurrent, il faut empêcher que deux processus se retrouvent simultanément dans leurs sections critiques.

En Java, il est très simple d'empêcher que deux threads exécutent en même temps une même méthode. Pour cela, il suffit d'utiliser le modificateur synchronized sur la méthode en question.

Le mot clé synchronized signifie qu'un thread a besoin d'une clé pour accéder au code synchronisé.

Pour protéger vos données, vous devez donc protéger les méthodes qui ont une action sur ces données.

Tout objet possède un vérrou, et ce vérrou n'a qu'une clé.

Lorsqu'un thread veut entrer dans une méthode synchronisée, il doit obtenir la clé de l'objet. Si la clé est disponible, i.e. aucun autre thread ne la détient, le thread prend la clé et entre dans la méthode. Dès lors le thread gardera la clé tant qu'il n'aura pas terminé la méthode synchronisée.

Si la clé de l'objet n'est pas disponible, aucun thread ne peut entrer dans une méthode synchronisée de cet objet.

Pourquoi ne pas tout synchroniser ?

Exemple sans synchronized

esil

```
class TestSync implements Runnable {
  private int solde;

  public void run() {
    for (int i = 0; i < 50; i++) {
      incrementer();
      System.out.println("le solde est de : " + solde);
    } }
  public void incrementer () {
    int i = solde;
    solde = i + 1;
  } }

public class TestTestSync {
  public static void main (String [] args) {
    TestSync tache = new TestSync();
    Thread a = new Thread (tache);
    Thread b = new Thread (tache);
    a.start();
    b.start();
  } }
```

Exemple avec synchronized

esil

```
class TestSync implements Runnable {
  private int solde;

  public void run() {
    for (int i = 0; i < 50; i++) {
      incrementer();
      System.out.println("le solde est de : " + solde);
    } }
  public synchronized void incrementer () {
    int i = solde;
    solde = i + 1;
  } }

public class TestTestSync {
  public static void main (String [] args) {
    TestSync tache = new TestSync();
    Thread a = new Thread (tache);
    Thread b = new Thread (tache);
    a.start();
    b.start();
  } }
```

voir complément de cours.