Applications Mobiles

II - Java Multi-thread : Synchronisation

- - - - - - - - - -

- - - - - - - - - -

ENSMA A3-S5 - période B

2023-2024

M. Richard
richardm@ensma.fr

Sommaire

Introduction

↪ Contexte

- le partage de ressources entre threads est une situation (très) fréquente
- normal dans une application ≪ concurrente ≫ → on ne cherche donc pas à l'éviter
- cette situation est néanmoins source des plus fréquentes erreurs
  - blocages, erreurs d'exécution, inconsistances des données, ...
- c'est le concept de section critique
  - section de code au sein duquel est réalisé au moins un accès à une ressource partagée et qui ne peut être exécuté par plus d'un thread à la fois.

↪ Solution :

- utiliser les différents mécanismes de synchronisation offert par le langage
- deux mécanismes de base sont présentés dans ce chapitre :
  - la synchronisation par le mot clé `synchronized`
  - l'interface `Lock` et ses différentes implémentations

Outil `synchronized`

Principe de l'utilisation de `synchronized` :

- méthode la plus basique en Java pour protéger une section de code
- un seul thread en exécution pourra exécuter une méthode qualifiée de `synchronized`
- si un autre thread tente d'accéder à cette méthode ou à tout autre également qualifiée de synchronized, il sera suspendu jusqu'à ce que l'autre thread termine l'exécution de sa méthode.
- Ainsi, toute méthode `synchronized` exécuté par un Thread correspondra à une section critique pour celui-ci.
- Attention :
  - si la méthode `synchronized` est une méthode statique, un seul thread pourra exécuter cette méthode. En revanche un autre thread pourra exécuter une autre méthode `synchronized` et non statique...

Type de communication :

- Cette méthode permet donc de réaliser une communication asynchrone entre différents Thread (principe du module de données).
- Ne permet pas de résoudre totalement le problème du *Lecteur/Ecrivain* puisque ne permet pas d'effectuer des lectures simultanées...

Exemple : le problème ...

- On modélise un compte bancaire sur lequel un thread peut déposer de l'argent et un autre thread peut en retirer. Si l'on fait autant d'opération de retrait et de dépôt, le solde du compte devrait donc rester le même...
- Codage naïf du compte :

```
1 public class Compte {
2   private int solde;
3   public Compte(int v){
4     solde = v;
5   }
6   public void setSolde(int v){
7     solde = v;
8   }
9   public int getSolde(){
10    return solde;
11  }
12 }
```

```
1 public void depotArgent(int v){
2   int tmp = solde;
3   try {
4     Thread.sleep(10);
5   } catch (InterruptedException e) {
6     e.printStackTrace(); }
7   tmp=tmp+v;
8   solde = tmp;
9 }
10 public void retraitArgent(int v){
11   int tmp = solde;
12   try {
13     Thread.sleep(10);
14   } catch (InterruptedException e) {
15     e.printStackTrace(); }
16   tmp=tmp-v;
17   solde = tmp;
18 }
```

Exemple : le problème ...

- Codage naïf du compte

```
1 public class Depot extends Thread {
2   private Compte monCompte;
3   public Depot(Compte c) {
4     monCompte = c;
5   }
6   @Override
7   public void run() {
8     for (int i=0 ;i<100;i++){
9       monCompte.depotArgent(1000);
10    }
11  }
12 }
```

```
1 public class Retrait extends Thread{
2   private Compte monCompte;
3   public Retrait(Compte c){
4     monCompte = c;
5   }
6   @Override
7   public void run() {
8     for (int i=0; i<100; i++){
9       monCompte.retraitArgent(1000);
10    }
11  }
12 }
```

Exemple : le problème ...

- Codage naïf du compte

```
1 public class Start {
2   public static void main(String[] args) {
3     Compte cpt = new Compte(1000);
4     Depot thDep = new Depot(cpt);
5     Retrait thRet = new Retrait(cpt);
6     System.out.println("Valeur compte Debut : " + cpt.getSolde());
7     thDep.start();
8     thRet.start();
9     try {
10      thDep.join();
11      thRet.join();
12    } catch (InterruptedException e) {
13      e.printStackTrace();
14    }
15    System.out.println("Valeur compte Fin : " + cpt.getSolde());
16  }
17 }
```

- On obtient par exemple lors d'une exécution :

```
Valeur compte Debut : 1000
Valeur compte Fin : -39000
```

  - clairement, il y a un souci...

Exemple : 2 solutions

- synchronized sur les *méthodes*

```
1   public synchronized void depotArgent(int v){
2     int tmp = solde;
3     ....
4
5     public synchronized void retraitArgent(int v){
6       int tmp = solde;
7       ....
```

- synchronized sur *l'instance* de Compte dans les classes Retrait et Depot

```
1   @Override
2   public void run() {
3     for (int i=0; i<100; i++){
4       synchronized (monCompte) {
5         monCompte.retraitArgent(1000);
6       }
7     }
8   }
```

  - Dans les deux cas on obtient :

```
Valeur compte Debut : 1000
Valeur compte Fin : 1000
```

Problématique (Rappel) :

- Le problème du *producteur/consommateur* est un problème récurent dans la programmation concurrente. Il s'agit d'un buffer (zone de stockage) limité en taille. 1 ou plusieurs Thread produise·nt des données dans le buffer et 1 ou plusieurs Thread consomment les données de celui-ci.
  Lorsque le buffer est plein, le·s producteur·s ne peut/peuvent plus produire ; de même lorsque le buffer est vide le·s consommateur·s ne peut/peuvent plus consommer.
- Deux types de communications sont donc mises en jeu dans ce problème :
  - une communication asynchrone pour l'accès aux données
  - une synchronisation permettant de gérer le blocage/déblocage des threads lorsque le buffer est plein/vide.

Solution :

- Java propose la notion de condition associée à un bloc protégé par `synchronized`
- On dispose pour cela des méthodes `wait`, `notify` et `notifyAll`.
  - wait : si un thread exécute cette méthode, la JVM endort le thread et libère l'instance contenant le bloc protégé par `synchronized` permettant ainsi à un autre thread d'exécuter une autre méthode protégée par `synchronized`
    - Attention, si un thread exécute un `wait` en dehors d'une section protégée par `synchronized`, une exception est générée.
  - `notify` et `notifyAll` : ces méthodes, si utilisées dans un code au sein de la même instance contenant le `wait` permet de réveiller le·s Thread précédemment endormi·s.

Exemple :

- La classe buffer :

```
1 public class MonBuffer {
2   private int tailleMax;
3   private List<String> zoneStock;
4   public MonBuffer(int taille){
5     tailleMax = taille;
6     zoneStock = new LinkedList<String>();}
7   public synchronized void deposeData(String data){
8     while(zoneStock.size()==tailleMax){
9       try { wait();
10      } catch (InterruptedException e) {
11        e.printStackTrace(); }}
12    zoneStock.add(data);
13    System.out.println("Ajout de : " +data);
14    notifyAll();}
15  public synchronized void consommeData(){
16    while(zoneStock.size()==0){
17      try { wait();
18      } catch (InterruptedException e) {
19        e.printStackTrace();}}
20    String data = zoneStock.remove(zoneStock.size()-1);
21    System.out.println("Consommé : " + data);
22    notifyAll(); }
23 }
```

- La classe Producteur

```
1 public class Producteur extends Thread {
2   private MonBuffer stock;
3   public Producteur(MonBuffer s){
4     stock = s;
5   }
6   @Override
7   public void run() {
8     for(int i=0;i<100;i++){
9       stock.deposeData("Depot "+i); }
10  }
11 }
```

- La classe Consommateur

```
1 public class Consommateur extends Thread {
2   private MonBuffer stock;
3   public Consommateur(MonBuffer s){
4     stock = s;
5   }
6   @Override
7   public void run() {
8     for(int i=0;i<100;i++){
9       stock.consommeData(); }
10  }
11 }
```

- Le main :

```
1 public class Start {
2   public static void main(String[] args) {
3     MonBuffer zone = new MonBuffer(25);
4     Consommateur conso = new Consommateur(zone);
5     Producteur prod = new Producteur(zone);
6     prod.start();
7     conso.start();
8   }
9 }
```

- Et en résultat :

```
Ajout de : Depot 0
Ajout de : Depot 1
Ajout de : Depot 2
Consommé : Depot 2
Consommé : Depot 1
Consommé : Depot 0
Ajout de : Depot 3
Ajout de : Depot 4
Ajout de : Depot 5
Ajout de : Depot 6
Consommé : Depot 6
Consommé : Depot 5
Consommé : Depot 4
...
```

Outil `Lock`

État des lieux :

- L'outil `synchronized` permet de répondre à un certain nombre de problématiques de manière assez simple.
- Néanmoins, ce concept possède un certain nombre de lacunes et défauts :
  - perte de performance du code
  - problème de l'accès de *n* thread en lecture simultanée et 1 seul thread en écriture
  - (très) difficile à mettre en œuvre lorsque le nombre de threads augmente
  - ne permet pas de gérer finement tous les problèmes de communication entre thread.

Nouveau mécanisme de synchronisation (à partir du JDK 1.5) : l'interface `Lock`

- Plus puissant et flexible que `synchronised`.
- Basée sur une interface (`Lock`) et sur un ensemble de classes l'implémentant (`ReentrantLock`, ...).
- Ajout de nouvelles fonctionnalités (`tryLock()`, `...`)
- Permet de distinguer les accès en lecture des accès en écriture.
- Plus performant en termes d'exécution

Fonctionnement : on souhaite ici garantir que lorsqu'un thread exécute une section critique, il soit le seul

- Utilisation d'une instance de la classe `ReentrantLock` qui implémente l'interface `Lock`
- Demande d'obtention du verrou à l'aide la méthode `lock()` sur l'instance, au début de la section critique :
  - Si aucun autre thread ne détient le verrou, il l'obtient et continue son exécution
  - Si un autre thread possède déjà le verrou, il est endormi (dans une file d'attente)
- Appel de la méthode `unlock()` à la fin de la section critique permettant de relâcher le verrou.
  - Remarque : attention lors de l'emploi de bloc `try ... catch` ; ne pas oublier de rendre le verrou dans la clause `finally`

Exemple : simulation d'une file d'impression

- Le processus souhaitant imprimer :

```
1 public class TacheImp extends Thread{
2   private FileImp maFile;
3   public TacheImp(FileImp f, String nomTh){
4     super(nomTh); maFile = f;
5   }
6   @Override
7   public void run(){
8     System.out.println(Thread.currentThread().getName() +
9     " : demande d'impression ...");
10    maFile.impressionDoc();
11    System.out.println(Thread.currentThread().getName() +
12    " : impresion terminée");
13  }
14 }
```

Exemple : simulation d'une file d'impression (suite)

- La file d'impression :

```
1 public class FileImp {
2   private final Lock leVerrou = new ReentrantLock();
3   public FileImp(){}
4   public void impressionDoc(){
5     leVerrou.lock();
6     try{
7       Long duree = (long)(Math.random()*1000);
8       System.out.println("Impression du travail du thread " +
9       Thread.currentThread().getName() + ": ... " + duree/100);
10      Thread.sleep(duree);
11    }catch(InterruptedException e){
12      e.printStackTrace();
13    }finally {
14      leVerrou.unlock();
15    }
16  }
17 }
```

- Le main :

```
1 public class Start {
2   public static void main(String[] args) {
3     FileImp file = new FileImp();
4     TacheImp[] tabTh = new TacheImp[10];
5     for(int i=0;i<5;i++){
6       tabTh[i] = new TacheImp(file,"Thread n°"+i);}
7     for(int i=0;i<5;i++){
8       tabTh[i].start(); }
9   }
10 }
```

- Et le résultat :

```
Thread n°4 : demande d'impression ...
Thread n°0 : demande d'impression ...
Thread n°3 : demande d'impression ...
Thread n°2 : demande d'impression ...
Thread n°1 : demande d'impression ...
Impression du travail du thread Thread n°4: ... 6
Thread n°4 : impresion terminée
Impression du travail du thread Thread n°0: ... 1
Thread n°0 : impresion terminée
Impression du travail du thread Thread n°3: ... 2
Thread n°3 : impresion terminée
Impression du travail du thread Thread n°2: ... 9
Thread n°2 : impresion terminée
Impression du travail du thread Thread n°1: ... 7
Thread n°1 : impresion terminée
```

- Une ressource est partagée entre plusieurs Thread accédant à la ressource soit en lecture soit en écriture.
- On autorise la lecture de manière simultanée (i.e. la lecture n'est pas bloquante), alors que l'écriture doit se faire de manière unique (aucun autre Thread ne doit accéder à la ressource durant l'écriture).
  - l'utilisation de `synchronize` ne permet pas la résolution de ce problème pourtant très courant dans la programmation concurrente.
- il s'agit d'une *communication asynchrone* entre *n* thread.

La solution : l'interface `ReadWriteLock` et son unique implémentation, la classe `ReentrantReadWriteLock`

- Cette interface propose deux méthodes supplémentaires : `readLock()` et `writeLock()`
- Comme leur nom l'indique, elle permette d'obtenir la ressource pour une opération de lecture ou pour une opération d'écriture. C'est au programmeur d'utiliser ces deux méthodes à bon escient.
  Précisément, les Thread effectuant un accès en lecture doivent utiliser la méthode `readLock()` pour obtenir le verrou, et ceux souhaitant faire une opération d'écriture, la méthode `writeLock()`.
  - ces deux méthodes donnent accès aux méthodes `lock`, `unlock()` et `tryLock()` pour obtenir, relâcher ou tester le verrou.
  - Attention, le fait d'obtenir le verrou par la méthode `readLock()` n'interdit pas de faire une opération d'écriture sur la ressource...

- la classe Livre :

```
1 public class Livre {
2   private String ligneLivre1;
3   private String ligneLivre2;
4   private ReadWriteLock monVerrou;
5   public Livre(){
6     ligneLivre1 = "lalala";
7     ligneLivre2 = "lilili";
8     monVerrou = new ReentrantReadWriteLock(); }
9   public String getLigne1(){
10    monVerrou.readLock().lock();
11    String tmp = ligneLivre1;
12    monVerrou.readLock().unlock();
13    return tmp; }
14  public String getLigne2(){
15    monVerrou.readLock().lock();
16    String tmp = ligneLivre2;
17    monVerrou.readLock().unlock();
18    return tmp; }
19  public void setLignes(String li1, String li2){
20    monVerrou.writeLock().lock();
21    System.out.println("...Ecriture du livre...");
22    ligneLivre1 = li1;
23    ligneLivre2 = li2;
24    System.out.println("...Fin Ecriture du livre...");
25    monVerrou.writeLock().unlock(); }
26 }
```

Exemple : deux chaînes pouvant être lues indépendamment, mais devant être modifiées simultanément

- la classe Lecteur :

```
1  public class Lecteur extends Thread {
2    private Livre monLivre;
3    public Lecteur(Livre li){
4      monLivre = li;
5    }
6    @Override
7    public void run() {
8      for(int i=0;i<3;i++){
9        System.out.println(Thread.currentThread().getName() + "-" + i + " ->ligne 1:" +
               monLivre.getLigne1());
10       System.out.println(Thread.currentThread().getName() + "-" + i + " ->ligne 2:" +
               monLivre.getLigne2());
11       try {
12         sleep(3);
13       } catch (InterruptedException e) {
14         e.printStackTrace(); }
15     }
16   }
17 }
```

Exemple : deux chaînes pouvant être lues indépendamment, mais devant être modifiées simultanément

- la classe Ecrivain :

```
1  public class Ecrivain extends Thread {
2    private Livre monLivre;
3    public Ecrivain(Livre li) {
4      monLivre = li;}
5    @Override
6    public void run() {
7      for (int i = 0; i < 3; i++) {
8        System.out.println(Thread.currentThread().getName() + "Modif du livre ...");
9        monLivre.setLignes("tatata"+i, "tititi"+i);
10       System.out.println(Thread.currentThread().getName() + "Modif terminée");
11       try {
12         sleep(3);
13       } catch (InterruptedException e) {
14         e.printStackTrace();}
15     }}
16 }
```

- En exécutant 5 lecteurs et 1 écrivain, on obtient :

```
...
Thread-4-0 ->ligne 1:lalala
Thread-4-0 ->ligne 2:lilili
Thread-0-0 ->ligne 1:lalala
Thread-5Modif du livre ...
Thread-0-0 ->ligne 2:lilili
...Ecriture du livre...
```

```
...Fin Ecriture du livre...
Thread-2-0 ->ligne 2:lilili
Thread-3-0 ->ligne 2:lilili
Thread-5Modif terminée
Thread-4-1 ->ligne 1:tatata0
Thread-3-1 ->ligne 1:tatata0
....
```

Définition & Fonctionnement :

- un verrou (i.e. toute instance de type Lock ou dérivée) peut être associé à *une* ou *plusieurs* condition
- l'objectif d'une condition est de proposer les méthodes nécessaires pour endormir ou réveiller un ou plusieurs Thread sur cette condition.
  - autorise ainsi l'utilisation des méthodes await(), signal() ou signallAll()
  - une liste d'attente est donc associée à chaque condition
- une condition est modélisée par l'interface Condition
  - une Condition est créée à l'aide de la méthode newCondition() définie dans l'interface Lock ou dérivée.
- méthode await() : attention, lorsqu'un Thread exécute cette méthode, sur une Condition, il relâche le verrou auquel est associée la Condition. Lorsque ce dernier est réveillé, par la méthode signal() ou signalAll(), il doit de nouveau acquérir le verrou.

Exemple : le producteur/consommateur avec des `Condition`

- Modification de la classe `MonBuffer` : attribut et constructeur

```
1 public class MonBuffer2 {
2   private int tailleMax;
3   private List<String> zoneStock;
4   private ReentrantLock monVerrou;
5   private Condition plein, vide;
6
7   public MonBuffer2(int taille){
8     tailleMax = taille;
9     zoneStock = new LinkedList<String>();
10    monVerrou = new ReentrantLock();
11    plein = monVerrou.newCondition();
12    vide = monVerrou.newCondition();
13  }
14  ...
15 }
```

Exemple : le producteur/consommateur avec des `Condition`

- Modification des méthodes :

```
1  ...
2  public void deposeData(String data) {
3    monVerrou.lock();
4    try {
5      while (zoneStock.size() == tailleMax) {
6        vide.await(); }
7      zoneStock.add(data);
8      System.out.println("Ajout de : " + data);
9      plein.signalAll();
10   } catch (InterruptedException e) {
11     e.printStackTrace(); } finally {
12     monVerrou.unlock();}
13 }
14 public void consommeData() {
15   monVerrou.lock();
16   try {
17     while (zoneStock.size() == 0) {
18       plein.await(); }
19     String data = zoneStock.remove(zoneStock.size() - 1);
20     System.out.println("Consommé : " + data);
21     vide.signalAll();
22   } catch (InterruptedException e) {
23     e.printStackTrace(); } finally {
24     monVerrou.unlock(); }
25 }
```

Exemple : le producteur/consommateur avec des `Condition`

- Résultat :

```
Ajout de : Depot 6
Ajout de : Depot 7
Consommé : Depot 7
Ajout de : Depot 8
Ajout de : Depot 9
Consommé : Depot 9
Ajout de : Depot 10
Ajout de : Depot 11
Ajout de : Depot 12
Consommé : Depot 12
Ajout de : Depot 13
Ajout de : Depot 14
Consommé : Depot 14
....
```

```
...
Ajout de : Depot 97
Consommé : Depot 97
Ajout de : Depot 98
Consommé : Depot 98
Ajout de : Depot 99
Consommé : Depot 99
Consommé : Depot 40
Consommé : Depot 39
Consommé : Depot 37
Consommé : Depot 35
Consommé : Depot 33
...
```

- Remarque : il existe plusieurs surcharges de la méthode `await` :
  - `await(long time, TimeUnit unit)` : permet de spécifier un timeout. le paramètre `unit` peut être l'une des valeurs `DAYS`, `HOURS`, `MICROSECONDS`, `MILLISECONDS`, `MINUTES`, `NANOSECONDS`,`SECONDS`
  - `awaitUntil(Date date)` : comme précédemment, mais en spécifiant une échéance plutôt qu'un délai.
  - `awaitUninterruptibly()` : idem que `await` mais ne peut-être interrompu.

Autres outils. . .

↪ Généralités

Depuis la version du JDK 1.5 et dans les suivantes un certain nombre d'outils évolués sont apparus :

- `Semaphore` : outil de base dans la programmation concurrente. Si bien utilisé, permet de régler la majorité de problème de synchronisation/communication se posant dans ce contexte.
- `CountDownLatch` : outil permettant à un thread d'attendre la terminaison de plusieurs opérations
- `CyclicBarrier` : outil permettant la synchronisation de plusieurs threads en un point
- `Phaser` : mécanisme permettant de contrôler des Thread assurant une opération divisée en phase. Tous les Thread doivent avoir terminé une phase afin de pouvoir commencer la phase suivante.
- `Exchanger` :outil permettant d'implémenter un point de synchronisation avec échange de données entre deux Thread.

Définition :

- Un sémaphore est un compteur permettant de contrôler l'accès à une ou plusieurs ressources partagées.
- Java propose une classe `Semaphore` modélisant cet outil incontournable
- Plusieurs méthodes sont fournies par cette classe :
  - `Semaphore(n)`,`Semaphore(n,b)` : constructeur ; il est possible d'utiliser un constructeur avec un paramètre d'initialisation du compteur, ou encore avec ce dernier paramètre et un booléen indiquant le comportement de choix du thread à réveiller (si true, mode équitable)
  - `acquire()` : permet d'acquérir le sémaphore
  - `release()` : permet de relâcher le sémaphore
  - `tryAcquire()` : permet de tester l'acquisition du sémaphore

Exemple : La file d'impression – version sémaphores

- La classe

```
1   private final Semaphore semFile;
2   public FileImp2(){
3     semFile = new Semaphore(1); }
4   public void impressionDoc(){
5     try{
6       semFile.acquire();
7       Long duree = (long)(Math.random()*1000);
8       System.out.println("Impression du travail du thread " +
9       Thread.currentThread().getName() + ": ... " + duree/100);
10      Thread.sleep(duree);
11    }catch(InterruptedException e){
12      e.printStackTrace();  } finally {
13      semFile.release();}
14  }
15 }
```

Exemple : La file d'impression – version sémaphores

- Les classes `Start` et `TacheImp` restent les mêmes
- Le résultat de l'exécution :

```
Thread n°2 : demande d'impression ...
Thread n°0 : demande d'impression ...
Thread n°3 : demande d'impression ...
Thread n°4 : demande d'impression ...
Thread n°1 : demande d'impression ...
Impression du travail du thread Thread n°2: ... 4
Thread n°2 : impresion terminée
Impression du travail du thread Thread n°0: ... 3
Thread n°0 : impresion terminée
Impression du travail du thread Thread n°3: ... 1
Thread n°3 : impresion terminée
Impression du travail du thread Thread n°4: ... 6
Thread n°4 : impresion terminée
Impression du travail du thread Thread n°1: ... 2
Thread n°1 : impresion terminée
```

Fonctionnement de l'outil `CountDownLatch` :

- Initialisation de l'instance avec le nombre n d'opérations à attendre
- Lorsque le thread exécute la méthode `await`, il est mis en attente de la terminaison des n opérations
- Lorsqu'une opération se termine elle doit appeler la méthode `countDown()` permettant de décrémenter la valeur n. Lorsque la valeur atteint 0, le thread est débloqué.
- Remarque :
  - la valeur du compteur ne peut être changé ; elle est fixée lors de l'instanciation
  - une instance ne peut donc servir qu'une seule fois
  - ce mécanisme n'assure en aucun cas la protection d'une ressource partagée. Il permet donc d'implémenter une synchronisation $1/n$.

Exemple : initialisation de connexions multiples

- Un Thread devant réaliser une opération de calcul doit attendre la connexion aux n bases de données dont il a besoin pour son calcul

Exemple : initialisation de connexions multiples

- La classe FaireCalcul :

```
1 public class FaireCalcul extends Thread {
2   private CountDownLatch ctrl;
3
4   public FaireCalcul(int nbconnex) {
5     ctrl = new CountDownLatch(nbconnex);
6   }
7   public void connexionOK(String mess){
8     System.out.println("Connexion : " + mess);
9     ctrl.countDown();
10  }
11  @Override
12  public void run() {
13    System.out.println("Initialisation du calcul : Attente des "
14    + ctrl.getCount() + " connexions aux bases ........");
15    try {
16      ctrl.await();
17      System.out.println("Toutes les connexions sont ok : Début du calcul ....");
18    } catch (InterruptedException e) {
19      e.printStackTrace(); }
20  }
21 }
```

Exemple : initialisation de connexions multiples

- La classe DataBaseConnect :

```
1 public class DataBaseConnect extends Thread {
2   private FaireCalcul calcul;
3   private String nomConnex;
4
5   public DataBaseConnect(FaireCalcul fc, String nom) {
6     calcul = fc;
7     nomConnex = nom;
8   }
9   @Override
10  public void run() {
11    long duration = (long) (Math.random() * 10);
12    try {
13      TimeUnit.SECONDS.sleep(duration);
14    } catch (InterruptedException e) {
15      e.printStackTrace(); }
16    calcul.connexionOK("Connexion à la DB " + nomConnex + "-> OK");
17  }
18 }
```

Exemple : initialisation de connexions multiples

- La classe Start

```
1 public class Start {
2   public static void main(String[] args) {
3     FaireCalcul fc = new FaireCalcul(5);
4     fc.start();
5     for(int i=0;i<5;i++){
6       DataBaseConnect db = new DataBaseConnect(fc, "DatBase"+i);
7       db.start();
8     }
9   }
10 }
```

- Le résultat :

```
Initialisation du calcul : Attente des 5 connexions aux bases ........
Connexion : Connexion à la DB DatBase4-> OK
Connexion : Connexion à la DB DatBase2-> OK
Connexion : Connexion à la DB DatBase1-> OK
Connexion : Connexion à la DB DatBase3-> OK
Connexion : Connexion à la DB DatBase0-> OK
Toutes les connexions sont ok : Début du calcul ....
```

Fonctionnement de l'outil `Exchanger`

- Permet de réaliser une communication synchrone (i.e. synchronisation + échange de données) entre **2** Thread
- la classe `Exchanger` qui donne accès à la méthode `exchange()`
  - Cette méthode est bloquante : le premier thread à exécuter cette méthode est endormi jusqu'à ce que l'autre Thread l'exécute également.
  - Une fois l'échange de données effectué, chacun des 2 threads reprend le cours de son exécution
  - Remarque : la classe `Exchanger` propose également une méthode `exchange()` prenant en paramètre une durée et une unité de temps, permettant de réaliser une attente bornée sur cette méthode.

Exemple : Alimentation indépendante de rail d'un circuit ferroviaire

- La classe Message modélisant le message entre les deux threads

```
1 public class Message {
2   private int noRail;
3   private boolean alimRail;
4   public Message(){
5     noRail=1; alimRail=false;
6   }
7   public int getNoRail(){
8     return noRail; }
9   public boolean getAlimRail(){
10    return alimRail; }
11  public void setMess(int num,boolean val){
12    noRail = num;
13    alimRail = val;
14  }
15 }
```

Exemple : Alimentation indépendante de rail d'un circuit ferroviaire

- La classe GestionRail

```
1 public class GestRail extends Thread {
2   private Message monMess;
3   private Exchanger<Message> echMess;
4
5   public GestRail(Message m, Exchanger<Message> exm) {
6     monMess = m;
7     echMess = exm;
8   }
9   @Override
10  public void run() {
11    for(int i=0;i<10;i++){
12      System.out.println("Préparation du message ...");
13      try {
14        sleep(500);
15        monMess.setMess(i, true);
16        System.out.println("Envoi du message "+i);
17        monMess = echMess.exchange(monMess);
18      } catch (InterruptedException e) {
19        e.printStackTrace(); }
20    }
21  }
22 }
```

Exemple : Alimentation indépendante de rail d'un circuit ferroviaire

- La classe CmdRail

```
1  public class CmdRail extends Thread {
2    private Message monMess;
3    private Exchanger<Message> echMess;
4
5    public CmdRail(Message m, Exchanger<Message> exm) {
6      monMess = m;
7      echMess = exm;
8    }
9    @Override
10   public void run() {
11     for(int i=0;i<10;i++){
12       System.out.println("Attente commande ...");
13       try {
14         monMess = echMess.exchange(monMess);
15       } catch (InterruptedException e) {
16         e.printStackTrace();
17       }
18       System.out.println("Commande : " + monMess.getNoRail() + "," +
                 monMess.getAlimRail());
19       try {
20         sleep(500);
21       } catch (InterruptedException e) {
22         e.printStackTrace(); }
23     }
24   }
25 }
```

↪ L'Exchanger 5/5

Exemple : Alimentation indépendante de rail d'un circuit ferroviaire

- La classe Start

```
1 public class Start {
2   public static void main(String[] args) {
3     Message lemessage = new Message();
4     Exchanger<Message> ech = new Exchanger<Message>();
5     GestRail thGest = new GestRail(lemessage, ech);
6     CmdRail thCmd = new CmdRail(lemessage, ech);
7     thCmd.start();
8     thGest.start();
9   }
10 }
```

- Le résultat :

```
Préparation du message ...
Attente commande ...
Envoi du message 0
Préparation du message ...
Commande : 0,true
Attente commande ...
Envoi du message 1
Préparation du message ...
Commande : 1,true
Attente commande ...
Envoi du message 2
Préparation du message ...
...
...
Attente commande ...
Envoi du message 7
Préparation du message ...
Commande : 7,true
Attente commande ...
Envoi du message 8
Préparation du message ...
Commande : 8,true
Attente commande ...
Envoi du message 9
Commande : 9,true
```