La concurrence

Introduction

La programmation que nous avons abordée jusqu'à présent repose sur un seul **flux d'exécution**. Cette approche pose des limites dans certains cas :

- Lorsque le programme interagit avec le monde extérieur, il se retrouve souvent à attendre que ce monde extérieur réagisse ; par exemple, pour écrire un bout de fichier sur un disque ou encore pour obtenir une réponse d'un utilisateur.
- De plus en plus les ordinateurs sont équipés de plusieurs processeurs. Chaque processeur peut traiter son propre flux d'exécution, mais si le programme n'en possède qu'un, un seul processeur sera réellement utilisé et le programme n'utilise qu'une fraction de la puissance de la machine.
- Certains programmes s'expriment naturellement beaucoup mieux à l'aide de plusieurs flux d'exécution. Un serveur web pourra traiter chaque requête dans un flux indépendant des autres requêtes en cours de traitement.

Utiliser des processus du système d'exploitation pour résoudre ceci génèrerait notamment beaucoup de problèmes de collaboration entre eux car ils dont indépendants et gérés par le système d'exploitation.

Thread

En programmation, peu importe le langage, on peut procéder à l'exécution de différents codes dans différents flux d'exécution en même temps (ou à des moments différés) Un flux d'exécution est appelé un **thread** en informatique.

Un thread est une version allégée de **processus**. L'idée est qu'un seul processus exécutera directement plusieurs threads. L'avantage de cette approche est multiple :

- Les threads sont légers comparés aux processus : beaucoup plus rapides à démarrer, beaucoup moins couteux en mémoire.
- Les threads partagent le même espace d'adressage et peuvent donc se partager des données et du code. Donc le coût de communication entre les threads est relativement bas par rapport à la celui entre les processus.
- Le fait pour un processeur de passer de l'exécution d'un thread à un autre (context switch en anglais) est moins coûteux que pour les processus.

Thread en Java

Il y a deux manières de créer des threads en Java :

1. Implémenter l'interface `java.lang.Runnable`
2. Etendre la classe `java.lang.Thread`

Interface Runnable

```
public interface Runnable {

      void run();

}
```

1. Il faut écrire une classe qui implémente `Runnable` et donc la méthode `run()`. Cette méthode contiendra la logique du thread.
2. Il faut ensuite créer une instance de la classe `Thread` en passant au constructeur une référence vers une instance de notre classe qui implémente `Runnable`. A ce stade, le thread n'est pas encore exécuté.
3. La méthode `start()`[1] de l'instance de `Thread` permet de le démarrer. Un nouveau flux d'exécution est créé et la méthode `run()` de la classe y est exécutée. La méthode `start()` se termine immédiatement, par contre, le thread dans lequel elle a été appelée continue à s'exécuter en parallèle de celui nouvellement créé.
4. Le nouveau thread s'exécute jusqu'à la terminaison normale de la méthode `run()` ou bien jusqu'à ce qu'il y ait une exception non attrapée ou encore jusqu'à la fin du processus lui-même.

Exemple :

```
class MonRunnable implements Runnable {
	private String affiche;

	public MonRunnable(String affiche) {
		this.affiche = affiche;
	}

	@Override
	public void run() {
		try {
			Thread.sleep((long) (Math.random() * 10000));
		} catch (InterruptedException e) {
		}
		System.out.println(this.affiche);
	}
}

class ExempleMonRunnable {
	public static void main(String[] args) {
		Thread thread1 = new Thread(new MonRunnable("hello"));
		Thread thread2 = new Thread(new MonRunnable("world"));
		Thread thread3 = new Thread(new MonRunnable("what's"));
		Thread thread4 = new Thread(new MonRunnable("up"));
		thread1.start();
		thread2.start();
		thread3.start();
		thread4.start();
	}
}
```

Dans l'exemple ci-dessus, la méthode `Thread.sleep(long ms)` permet d'endormir un thread pendant un certain temps en millisecondes. Le programme affichera donc les 4 chaînes de caractères dans un ordre aléatoire.

[1] Attention, appeler la méthode `run()` au lieu de `start()` exécute le code du thread dans le flux d'exécution courant.

Classe Thread

1. Il faut écrire une classe qui étend `Thread` et redéfinit la méthode `run()`. Cette méthode contiendra la logique du thread.
2. Il faut ensuite créer une instance de notre classe. A ce stade le thread n'est pas encore exécuté.
3. La méthode `start()` permet de démarrer ce dernier. Un nouveau flux d'exécution est créé et la méthode `run()` de la classe y est exécutée. La méthode `start()` se termine immédiatement, par contre, le thread dans lequel elle a été appelée continue à s'exécuter en parallèle de celui nouvellement créé.
4. Le nouveau thread s'exécute jusqu'à la terminaison normale de la méthode `run()` ou bien jusqu'à ce qu'il y ait une exception non attrapée.

```
class MonThread extends Thread {
	private String affiche;
	public MonThread(String affiche) {
		this.affiche = affiche;
	}

	@Override
	public void run() {
		try {
			Thread.sleep((long) (Math.random() * 10000));
		} catch (InterruptedException e) {
		}
		System.out.println(this.affiche);
	}
}

class ExempleMonThread {
	public static void main(String[] args) {
		MonThread thread1 = new MonThread("hello");
		MonThread thread2 = new MonThread("world");
		MonThread thread3 = new MonThread("what's");
		MonThread thread4 = new MonThread("up");
		thread1.start();
		thread2.start();
		thread3.start();
		thread4.start();
	}
}
```

Cet exemple a le même comportement que l'exemple précédent.

Cycle de vie d'un thread Java

Soit une classe `MonThread` qui implémente `Runnable` ou étende `Thread`.

1. Lorsqu'on instancie `MonThread`, on n'a pas encore un nouveau thread Java. Par contre on a bien une instance d'une classe que l'on peut manipuler comme d'habitude (changer la valeur de ses attributs, utiliser ses méthodes, etc..).
2. Le thread est créé et démarre réellement par l'appel de la méthode `start()`.
3. Ce thread se termine lorsque la méthode `run()` se termine ou bien si une exception s'est échappée pendant son exécution. Certaines techniques permettent de forcer l'arrêt d'un thread de l'extérieur, confer ci-dessous.

4. Une fois le thread terminé, l'instance de `MonThread` continue à exister et pour peu que l'on ait une référence vers elle à partir d'un autre thread, on pourra donc continuer à l'utiliser pour par exemple récupérer des résultats.

Forcer un thread à s'arrêter à partir d'un autre thread

Un thread peut décider lui-même de s'arrêter relativement facilement, il suffit qu'il s'arrange pour terminer la méthode `run()` lorsqu'il doit s'arrêter :

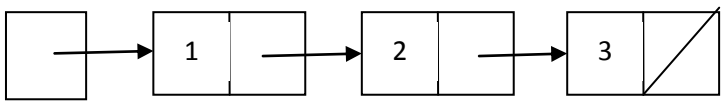
```
public class MonThread extends Thread {

	@Override
	public void run() {
		boolean ilFautContinuer = true;
		while(ilFautContinuer) {
			// faire qqch ici
		}
	}
}
```

On pourrait aussi jeter une exception qui ne sera pas attrapée pour interrompre le thread, mais ceci est déconseillé car entre autre cela laisse apparaître des messages d'erreur et des stack traces dans la console.

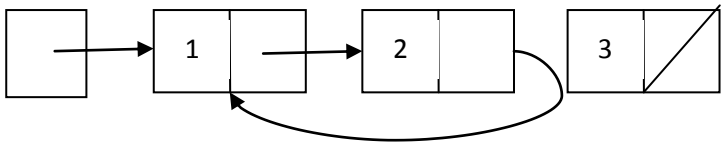
Par contre il est beaucoup plus délicat d'arrêter un thread de l'extérieur, c.-à-d. d'un autre thread. En effet, si on force un arrêt sauvage instantané d'un thread peu importe ce qu'il était en train de faire, on risque d'avoir de sérieux problèmes de corruption de la mémoire.

Admettons que deux threads utilisent une structure de données de type liste chaînée. Un thread demande à la liste de s'inverser, mais pendant que l'algorithme se déroule il est interrompu par l'autre thread. La liste est dans un état corrompu et on pourrait par exemple avoir une boucle infinie lorsque l'on demande la taille de la liste.

Liste chaînée originale



Liste chaînée en cours d'inversion



Si l'inversion est interrompue et que l'on demande la taille, le baladeur cyclera !

Pour des raisons historiques, la classe `Thread` est munie d'une méthode `stop()` qui permet d'arrêter brutalement comme ci-dessus. Comme nous venons de le voir, ce n'est pas une bonne manière d'arrêter un thread et **l'usage de cette méthode est donc interdit**.

Une manière possible de s'arrêter est donc :

```
public class MonThread extends Thread {
	private boolean ilFautContinuer = true;

	@Override
	public void run() {
		while(ilFautContinuer) {
			// faire qqch ici
		}
	}

	public void arret() {
		ilFautContinuer = false;
	}
}
```

Cette technique marche bien sauf si le `// faire qqch ici` prend beaucoup de temps. En effet dans ce cas il faudra attendre longtemps entre le moment où on demande l'arrêt du thread et le moment où il s'arrête réellement. Java fournit donc une technique permettant d'interrompre certaines opérations lentes afin d'éviter ce problème :

```
public class MonThread extends Thread {

	@Override
	public void run() {
		while(!isInterrupted()) {
			// faire qqch ici
		}
	}
}
```

Il existe un attribut `interrupt` au sein de la classe `Thread` qui vaut `false` par défaut. La méthode `isInterrupted()` est le getter de cet attribut. Un autre thread peut appeler la méthode `interrupt()` pour mettre l'attribut à `true`.

Cependant, si le thread interrompu était bloqué par un appel de certaines méthodes de Java qui sont connues pour avoir un temps d'exécution qui peut être long[2], alors l'attribut n'est pas mis à vrai mais à la place une exception `InterruptedException` est lancée dans le thread interrompu. Généralement la partie `catch` qui attrape cette exception remet l'interruption en appelant la méthode `interrupt()`.

```
public class MonThread extends Thread {

	@Override
	public void run() {
		while(!isInterrupted()) {
			try {
				Thread.sleep(100000);
			} catch (InterruptedException e) {
				interrupt();
			}
		}
```

[2] `wait()`, `wait(long)` ou `wait(long,int)` de la classe `Object`, ou bien par un appel de `join()`, `join(long)`, `join(long,int)`, `sleep(long)` ou `sleep(long,int)` de la classe `Thread`.

```
	}
}
```

Race Condition

Imaginons un système de votes utilisant une classe `Compteur` pour compter le nombre de votes pour un candidat, avec plusieurs threads lisant les votes en parallèle.

```
class Compteur {
	private int compteur = 0;

	public int getCompteur() {
		return compteur;
	}

	public void setCompteur(int v) {
		compteur = v;
	}
}

class VoteurThread extends Thread {
	private static Compteur compteur = new Compteur();

	public void run() {
		for (int i = 0; i < 100; i++) {
			int old = compteur.getCompteur();
			try {
				Thread.sleep(10); // dormir 10 msec
			} catch (InterruptedException e) {
				interrupt();
			}
			compteur.setCompteur(old + 1);
		}
	}

	public static void main(String[] args) {
		new VoteurThread().start();
		new VoteurThread().start();
		try {
			Thread.sleep(5000); // attendre suffisamment longtemps
		} catch (InterruptedException e) {
		}
		System.out.println(VoteurThread.compteur.get());
	}
}
```

Chaque exécution de la méthode `run()` augmente la variable `compteur` de 100. Cependant si on exécute deux threads simultanément le résultat n'est pas 200 ! Les deux threads se marchent, en effet, sur les pieds et on pourra avoir à un certain moment :

Thread A	Thread B
`int old = 10`	
	`int old = 10`
`Compteur = 10+1`	
	`Compteur = 10+1`

En anglais ce phénomène s'appelle **race condition** : il y a une course entre les deux threads et suivant la manière dont les exécutions s'entrelacent, on a des comportements indésirables.

Lorsqu'un bout de code n'assure plus son comportement correct s'il est appelé simultanément par plus d'un thread en parallèle, on dit qu'il n'est pas **thread-safe**. Si, au contraire, il continue à assurer son fonctionnement correct peu importe le nombre de threads, alors on dit qu'il est thread-safe.

Une méthode est dite thread-safe si elle garantit son fonctionnement quand on l'invoque sur la même instance à partir de plusieurs threads en parallèle. Une classe est dite thread-safe si le fonctionnement de toutes ses méthodes sont garanties quand on les invoque sur la même instance à partir de plusieurs threads en parallèle.

Le code de notre exemple n'est pas thread-safe. La source du problème est la variable statique partagée entre les deux threads. Cette variable est lue et écrite à des instants différents, en faisant la supposition que lorsqu'elle est écrite, elle n'a pas été modifiée ailleurs entretemps. Ceci est vrai pour une application avec un seul thread mais ne l'est plus dans notre exemple. Java propose plusieurs solutions à ce problème, en voici une : on marque les manipulations qui ne doivent être exécutées que par un seul thread à la fois.

Synchronized

L'idée est la suivante : chaque objet possède un jeton de **synchronisation** unique. Un bloc de code peut être marqué `synchronized` ce qui signifie que le thread exécutant ce code doit d'abord obtenir le jeton. Comme le jeton est unique lorsqu'un thread s'est approprié le jeton, tout autre thread voulant exécuter un bloc `synchronized` de la même instance devra d'abord attendre que le jeton soit libéré. Notons qu'un thread en train d'exécuter un bloc `synchronized` sur une instance pourra directement rentrer dans un autre bloc `synchronized` de la même instance puisqu'il en possède déjà le jeton. On appelle un bloc de code ainsi synchronisé une **section critique**.

```
Class Compteur {
	private int compteur = 0 ;

	public int getCompteur() {
		return compteur ;
	}

	public void setCompteur(int v) {
		compteur = v ;
	}
}

class VoteurThread extends Thread {
	private static Compteur compteur = new Compteur() ;

	public void run() {
		for (int i = 0 ; i < 100 ; i++) {
			synchronized (compteur) {
				int old = compteur.getCompteur() ;
				try {
					Thread.sleep(10) ; // dormir 10 msec
				} catch (InterruptedException e) {
					interrupt() ;
				}
				compteur.setCompteur(old + 1) ;
			}
		}
	}
```

```
	public static void main(String[] args) {
		new VoteurThread().start() ;
		new VoteurThread().start() ;
		try {
			Thread.sleep(5000) ; // attendre suffisamment longtemps
		} catch (InterruptedException e) {
		}
		System.out.println(VoteurThread.compteur.get()) ;
	}
}
```

Dans la version ci-dessus, le corps de la boucle est placé dans un **bloc de synchronisation** sur l'instance du compteur. Lorsqu'un thread rentre dans ce bloc, il prend donc le jeton sur l'instance compteur. L'autre thread devra donc attendre la fin de l'exécution de ce bloc avant de pouvoir s'approprier le jeton et y rentrer à son tour. Il n'y aura plus de modification de la valeur du `compteur` entre le `get()` et le `set()` ; ce programme est thread-safe.

Thread A	Thread B
Prend le jeton de compteur	
`int old = 10`	
	Tente de prendre le jeton du compteur mais bloque
`Compteur = 10+1`	
Relâche le jeton de compteur	
	Prend le jeton de compteur
	`int old=11`
	`compteur=11+1`
	Relâche le jeton de compteur

Dead-lock

Avant d'aborder la bonne manière de synchroniser un code, il est important de réaliser les problèmes encourus.

Thread A	Thread B
Prend le jeton de l'objet1	
	Prend le jeton de l'objet2
Tente de prendre le jeton de l'objet2 et bloque	
	Tente de prendre le jeton de l'objet1 et bloque

Les deux threads finissent par bloquer indéfiniment et l'application cesse de fonctionner correctement. En anglais on appelle ceci un **dead-lock**. Ce genre de problème est excessivement difficile à débugger : on n'a pas une exception avec un stack trace permettant de tracer l'origine du problème. Le programme ne plante pas, il cesse juste de fonctionner. De plus le dead-lock peut être dû à une succession d'acquisition de jetons très complexes et qu'on ne peut changer arbitrairement sans provoquer d'autres dead-locks. Lorsqu'on synchronise du code il est donc vital de se tenir à une bonne pratique qui évite les dead-locks.

Heureusement, Coffman, en 1971, a démontré les conditions nécessaires et suffisantes à l'existence d'un dead-lock :

1. **Exclusion mutuelle** : il existe une ressource qui n'est utilisable que par un seul thread à la fois.
2. **Prendre et tenir** : celui qui prend la ressource la garde pendant qu'il demande des ressources supplémentaires.
3. **Pas de préemption** : seul celui qui prend la ressource peut la relâcher, il n'est pas possible de le faire d'une manière externe.
4. **Attente circulaire** : deux ou plusieurs threads forment une chaîne circulaire où chaque thread attend la ressource du thread suivant de la chaîne.

Regardons comment ces conditions peuvent être contrôlées ou pas avec le mécanisme de synchronisation de Java :

1. Exclusion mutuelle : il ne faut synchroniser que ce qui est nécessaire afin d'éviter les exclusions mutuelles inutiles. En particulier si deux threads n'utilisent pas de mémoire partagée il n'y a pas de raison de les synchroniser entre eux. En général on n'a pas de contrôle sur l'exclusion mutuelle en Java : certaines parties du code en ont besoin et on n'a donc pas le choix.
2. Prendre et tenir : on est dans cette situation lorsqu'un bloc `synchronized` sur une instance particulière essaie de rentrer dans un autre bloc `synchronized` d'une autre instance. On pourrait éviter complètement de prendre et tenir en synchronisant systématiquement sur la même instance (par exemple `Object.class`), mais cela aurait évidemment un impact non négligeable sur la performance, on serait vite dans une situation quasi mono-thread. En Java on n'a donc souvent pas d'autre choix que de prendre et tenir.
3. Pas de préemption : comme nous l'avons vu, une thread externe ne peut pas à proprement parler interrompre un autre thread (`Thread.stop()` étant interdit d'utilisation). Il n'y a donc pas de préemption en Java.
4. Attente circulaire : on a peu de contrôle sur les points précédents en Java, c'est donc sur ce point-ci qu'il faudra souvent se rattraper. La manière d'éviter les cycles est simple : on s'arrange pour prendre les ressources toujours dans le même **ordre** rendant ainsi impossible l'existence d'un cycle. Par exemple :
   - Thread 1 : synchronize A,B,C
   - Thread 2 : synchronize B,C
   - Thread 3 : synchronize A,C

   Si le thread 1 s'empare de A, le thread 3 devra attendre. Ensuite le thread 2 peut prendre B, ce qui bloque le thread 1 et laisse C au thread 2. Une fois qu'il a terminé, le thread 1 peut prendre B et C, terminer et le thread 3 pourra enfin s'exécuter.

Quand synchroniser une méthode ?

Manipulation de l'état intrinsèque

Lorsqu'une méthode manipule son état d'une manière non thread-safe, on devra la synchroniser sur `this`. En Java, on peut ajouter `synchronized` à la signature de la méthode afin que son bloc complet soit en fait `synchronized` sur `this`.

```
public class ExempleSynchronized {

	private int compteur=0;
	public void inc() {
		compteur++;
	}
	public int getCompteur() {
		return compteur;
	}
}
```

Ce code n'est pas thread-safe : `compteur++` souffre d'un race condition puisqu'il est équivalent à `compteur = compteur+1` et que nous avons vu qu'un tel code n'est pas thread-safe. Nous pourrons le corriger comme ceci :

```
public class ExempleSynchronized {

	private int compteur = 0;
	public void inc() {
		synchronized(this) {
			compteur++;
		}
	}
}
```

Qui peut aussi s'écrire comme ceci, les deux codes étant strictement équivalent :

```
public class ExempleSynchronized {

	private int compteur = 0;
	public synchronized void inc() {
		compteur++;
	}
}
```

Si on ajoute une méthode `getCompteur()`, il faut évidemment réfléchir aussi à ce qu'elle renvoie toujours la dernière valeur à jour du `compteur`. Deux façons de procéder :

Déclarer un champ volatile

Le champ `compteur` est déclaré `volatile`. Cette solution est plus "légère" que la synchronisation. Un champ déclaré `volatile` ne se comporte pas comme un champ normal.

Toute écriture d'un tel champ est appelée *lecture volatile*, et toute écriture est une *écriture volatile*. Toute lecture volatile d'un champ retourne la valeur fixée par la dernière écriture volatile du champ. Ceci s'applique systématiquement, indépendamment du thread dans lequel la lecture ou l'écriture ont eu lieu.

Déclarer un champ volatile garantit que la valeur de ce champ, vue de n'importe quel thread, est toujours à jour. Etant non synchronisé, l'accès à un champ volatile est plus performant que l'accès à un champ qui se ferait au travers d'un bloc synchronisé. Dans la pratique, déclarer un champ volatile fait que la machine Java ne stockera pas ce champ dans un registre, ce qui rendra les lectures de ce champ légèrement moins performantes.

Synchroniser les getters

La méthode `getCompteur()` est `synchronized`. Il faut donc obtenir le jeton également pour accéder à la valeur à tout moment.

```
public class ExempleSynchronized {

	private int compteur = 0;
	public synchronized void inc() {
		compteur++;
	}
	public synchronized int getCompteur() {
		return compteur;
	}
}
```

En général, on synchronise le(s) getter(s) afin de s'assurer que l'on n'accède aux attributs que lorsque l'on est sûr que l'état global de l'objet est consistant :

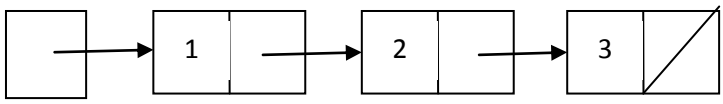
```
public class ExempleSynchronized {

	private int compteur = 0;
	private int decompteur = 0;
	public synchronized void inc() {
		compteur++;
		decompteur--;
	}
	public synchronized int getCompteur() {
		return compteur;
	}
	public synchronized getDecompteur() {
		return decompteur;
	}
}
```

Ici, il devient nécessaire de synchroniser les getters : un thread pourrait accéder aux deux valeurs des attributs alors qu'elles ne sont pas l'opposée l'une de l'autre perdant ainsi la cohérence de l'état global de l'instance. Ceci n'est pas l'unique raison pour laquelle il est nécessaire de synchroniser une méthode ; les structures de données manipulées le sont également.

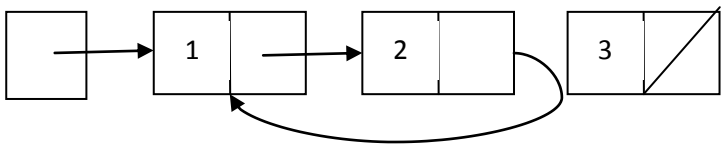
Structures de données non thread-safe

Rappelons-nous du problème d'interruption d'un thread : si on permet d'interrompre, à n'importe quel instant, un thread utilisant une liste chaînée en cours d'inversion alors on risque de la laisser dans un état corrompu la rendant inutilisable.

Liste chaînée originale



Liste chaînée en cours d'inversion



Si l'inversion est interrompue et que l'on demande la taille, le baladeur cyclera !

Le même problème se pose bien évidemment si on permet à un autre thread d'utiliser la liste chaînée alors qu'on est en train de l'inverser. Toute méthode utilisant une structure de données non thread-safe devra donc être synchronisée. Pour chaque structure de données native de Java, il est donc nécessaire de consulter la Javadoc pour savoir si elle est thread-safe ou pas. Par exemple `HashMap` n'est pas thread-safe alors que `Hashtable` l'est !

```
class ExempleHashtable {

	private Hashtable<String, Integer> hash = new Hashtable<String, Integer>();

	public void put(String key, Integer value) {
		hash.put(key, value);
	}

	public Integer get(String key) {
		return hash.get(key);
	}
}
```

Cette classe n'a pas besoin d'être synchronisée, elle peut l'être mais ce n'est pas une obligation.

```
class ExempleHashMap {

	private HashMap<String, Integer> hash = new HashMap<String, Integer>();

	public synchronized void put(String key, Integer value) {
		hash.put(key, value);
	}

	public synchronized Integer get(String key) {
		return hash.get(key);
	}
}
```

Cette classe devra obligatoirement être synchronisée !

Appels de méthode

Les structures de données natives de Java ne sont qu'un cas particulier d'un cas plus général : comment synchroniser une méthode qui en appelle une autre, en particulier si la méthode appelée appartient à une autre instance qui elle aussi a besoin d'être synchronisée. Dans ce cas en effet il y a plusieurs jetons de synchronisation à acquérir :

- Si on peut se contenter de prendre un jeton à la fois, alors on les prend successivement afin d'éviter la condition *prendre et tenir* de Coffman évitant ainsi les dead-locks :

```
class Compte {
   private double solde;

   public void versement(double montant, Compte c) {
         synchronized (this) {
               this.solde -= montant;
         }
         c.dépot(montant); // où dépôt est une méthode synchronized
   }

   public synchronized void dépot(double montant) {
         this.solde += montant;
   }

}
```

Dans cet exemple pour la méthode virement, il y a deux jetons à prendre : celui sur `this` et celui sur `c`. On peut cependant séparer les synchronisations et ainsi éviter les dead-locks.

- S'il est nécessaire de prendre plusieurs jetons à la fois, alors il faudra décider d'un ordre pour les prendre et respecter cet ordre **partout** :

```
public void transférer (Portefeuille p, Compte c){
      synchronized(c) {
          synchronized(p) {
              c.dépot(p.getContenu());
          }
      }
}
```

Pour la méthode transférer, on a décidé de d'abord synchroniser sur l'instance de `Compte`, et ensuite sur celle de `Portefeuille`. Dorénavant chaque fois que l'on devra synchroniser sur des instances de ces types, il faudra **toujours** synchroniser d'abord sur `Compte` et puis sur `Portefeuille`.

Mais que se passe-t-il s'il faut synchroniser sur des instances différentes du même type ? De nouveau il faudra trouver un ordre global et s'y tenir. Si on n'a pas un moyen d'obtenir un ordre facilement (par un attribut unique comparable par exemple), on peut utiliser la fonction `System.identityHashCode` :

```
public void transférer (Compte c1, Compte c2){
      Compte premier, second;
      if (System.identityHashCode(c1) < System.identityHashCode(c2)){
            premier = c1; second = c2;
      } else {
            premier = c2; second = c1;
      }
      synchronized(premier) {
          synchronized(second) {
              c1.dépot(c2.getSolde());
          }
      }
}
```

Appels de méthodes statiques

La technique de synchronisation pour les méthodes statiques est similaire à celle pour les méthodes non-statiques à un détail près : il n'y a pas de `this` pour les méthodes statiques. A la place on prendra donc l'attribut `.class` de la classe elle-même :

```
public class CompteTout {
      private static int compte = 0;

      public static void incr() {
            synchronized (CompteTout.class) {
                  compte++;
            }
      }

      public static synchronized void decr() {
            compte--;
      }
}
```

Remarquons qu'une méthode `static synchronized` est en fait une méthode statique dont le corps est `synchronized` sur le `.class`.

Attention aussi à l'erreur de distraction : une méthode non statique mais `synchronized` ne synchronisera pas sur le `.class`, juste sur le `this` :

```
public class CompteTout {
	private static int compte = 0;

	public static void incr() {
		synchronized (CompteTout.class) {
			compte++;
		}
	}

	public static synchronized void decr() {
		compte--;
	}

	public synchronized int getCompte() {
		return compte;
	}

}
```

Le code ci-dessus est incorrect !

```
public class CompteTout {

	private static int compte = 0;

	public static void incr() {
		synchronized (CompteTout.class) {
			compte++;
		}
	}

	public static synchronized void decr() {
		compte--;
	}

	public int getCompte() {
		synchronized (CompteTout.class) {
			return compte;
		}
	}
}
```

Ceci est bien correct. Notons que si `getCompte()` avait besoin de synchroniser aussi sur le `this`, il faudrait donc décider d'un ordre pour prendre les deux jetons (celui sur `this` et celui sur `.class`) et s'y tenir dans le reste du programme.

Technique avancée : synchronisation sur this et sur un attribut

Admettons qu'une méthode d'une classe `A` doit se synchroniser sur `this` et sur une instance de `B` retenue dans l'attribut `this.b`. Une solution possible pourrait être :

```
class A {
	private B b;
	public A(B b) {
		this.b=b;
	}
	public synchronized void maMethodeCompliquée() {
		if (this.b == null) throw new IllegalStateException();
		synchronized (this.b) {
			// code compliqué ici
		}
	}
}
```

On voit qu'on synchronise d'abord sur l'instance de `A`, puis sur celle de `B`. Mais que se passe-t-il si une méthode de `B` a un besoin similaire : se synchroniser sur `this` et sur une instance de `A` retenue dans son attribut `this.a` ? On est obligé de respecter l'ordre de synchronisation sinon on risque un dead-lock. Il faudra donc procéder par étape :

```
class B {
	private A a;

	public B(A a) {
		this.a = a;
	}

	public void maMethodeCompliquée() {
		A autre = null;
		do {
			synchronized (this) {
				if (this.a == null)
					throw new IllegalStateException();
				autre = a;
			}
			synchronized (autre) {
				synchronized (this) {
					if (autre == this.a) {
						// code compliqué ici
						return;
					}
				}
			}
		} while (true);
	}
}
```

L'attribut `this.a` pourrait changer si on ne synchronise pas sur `this`, on est donc obligé de commencer par cela. Mais bien sûr on ne peut pas à l'intérieur de ce bloc là aussi synchroniser sur `this.a`, sinon dead-lock ! On validera donc que l'attribut n'est pas `null` afin de pouvoir synchroniser dessus et on le stockera dans une variable locale. C'est sur base de cette variable locale que l'on synchronisera sur l'instance de `A` puis sur l'instance de `B` : en effet cette variable ne changera pas. Cependant l'attribut lui peut changer entre les deux blocs de synchronisation et il faudra donc tester si c'est le cas ou pas. S'il est resté le même, on pourra continuer le traitement et

terminer la méthode normalement, sinon il faudra réessayer avec la nouvelle valeur de l'attribut. Cette formule est bien sûr à adapter en fonction de la méthode.

Technique avancée : synchronisation d'une relation bidirectionnelle

Lorsqu'on a une relation bidirectionnelle, chaque instance a besoin d'être synchronisée sur son propre jeton. En général, on devra synchroniser toutes les méthodes, y compris les méthodes de minimum et maximum sur la multiplicité de la relation. Sinon on aurait un premier thread manipulant cette multiplicité et un second thread l'observant d'une manière non cohérente.

Parfois on sera obligé de se synchroniser sur les deux instances. Il faudra donc décider d'un ordre sur les classes, et s'y tenir. Comme le point précédent le montre, suivant les méthodes et le besoin de synchronisation il y a souvent un ordre qui est plus simple. Il faudra donc vérifier s'il y a un sens qui évite le cas difficile ci-dessus.

Attention aussi à synchroniser les paramètres des méthodes lorsque cela est nécessaire :

```
	public synchronized void ajouterCommande(Commande c) {
		Util.checkObject(c);
		synchronized (c) {
			// reste de la méthode
		}
	}
```

Technique avancée : synchronisation d'un singleton lazy en Java

Si vous implémentez un singleton d'une manière lazy à l'aide du code :

```
	public static Singleton getInstance() {
		if (instance == null)
			synchronized (Singleton.class) {
				if (instance == null)
					instance = new Singleton();
			}
		return instance;
	}
```

Ce code ne marche pas à cause de la manière dont le code Java est traduit en byte code, confer http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html. A la place, il faut écrire :

```
	public static synchronized Singleton getInstance() {
		if (instance == null)
			instance = new Singleton();
		return instance;
	}
```

Technique avancée : synchronisation des éléments d'une collection

Admettons qu'on doive synchroniser tous les éléments d'une collection. Vu la syntaxe du `synchronized`, on est bloqué. Par exemple imaginons une application de ventes en ligne qui a besoin de synchroniser sur tous les montants d'une commande pour en calculer le total :

```
	public synchronized double getTotalDeTout(){
		double total = 0;
		for (Commande c: commandes)
			synchronized (c) {
```

```
                        total += c.getMontant();
                }
                return total;
        }
```

Ceci ne marche pas : on synchronise bien sur chaque `Commande` individuellement, mais jamais sur toutes les commandes ensembles. Il se pourrait que la commande 1 change alors qu'on est déjà en train de sommer la commande 4 …

Java fournit des constructions qui permettent de gérer cela (théorie suivante).

Package java.util.concurrent.locks.Lock

On trouve dans ce package l'interface `Lock` implémentée par la classe `ReentrantLock`. Une instance d'un `ReentrantLock` contient un jeton de synchronisation et permet de le manipuler sur base d'appels de méthodes :

- `void lock()` : acquiert le lock, bloque jusqu'à sa disponibilité si nécessaire.
- `boolean tryLock()` : essaie d'acquérir le lock : s'il est disponible il est pris et la méthode retourne `true`, sinon il n'est pas pris et la méthode termine immédiatement en retournant `false`.
- `void unlock()` : libère le lock.

Il y a d'autres méthodes, mais aussi interfaces et classes dans ce package, confer la Javadoc.

Lorsqu'on souhaite synchroniser une classe non plus sur base de `synchronized`, mais sur base d'un `ReentrantLock`, il faut :

1. Créer un champ `private Lock lock = new ReentrantLock()`
2. Créer une méthode `public Lock getLock() {return lock ;}`
3. A chaque fois que l'on aurait utilisé `synchronized`, il faut le remplacer par le lock dans un bloc **`try {lock.lock() ;` … `} finally {lock.unlock() ;}`**:

```
class A {
        private B b;
        public A(B b) {
                this.b = b;
        }
        public synchronized void maMethodeCompliquée() {
                synchronized (this.b) {
                        // code compliqué ici
                }
        }
}
```

Devient :

```
class A {
        private B b;
        private Lock lock = new ReentrantLock();
        public A(B b) {
                this.b = b;
        }
        public Lock getLock() {
                return this.lock;
```

```
	}
	public void maMethodeCompliquée() {
		try {
			lock.lock();
			synchronized (this.b) {
				// code compliqué ici
			}
		} finally {
			lock.unlock();
		}
	}
}
```

Le bloc `try {lock.lock() ; … } finally {lock.unlock() ;}` est nécessaire pour garantir le bon relâchement du lock. Il est souvent possible d'obtenir le même résultat sans ce bloc try, mais un risque inutile de non relâchement existe. Finalement notons que lorsque la classe `B` voudra synchroniser sur `A`, elle devra aussi le faire via le lock de `A` :

```
class B {
	private A a;

	public B(A a) {
		this.a = a;
	}

	public void maMethodeCompliquée() {
		A autre = null;
		do {
			synchronized (this) {
				if (this.a == null)
					throw new IllegalStateException();
				autre = a;
			}
			try {
				autre.getLock().lock();
				synchronized (this) {
					if (autre == this.a) {
						// code compliqué ici
						return;
					}
				}
			} finally {
				autre.getLock().unlock();
			}
		} while (true);
	}
}
```

Dans cette version la classe `A` est synchronisée avec un lock tandis que la classe `B` est synchronisée par le `synchronized` classique. Les deux types de synchronisations peuvent se mélanger tant que l'on reste cohérent : les instances de `A` seront toujours synchronisées par des locks, celles de `B` toujours par `synchronized`. Nous pouvons aussi uniformiser en synchronisant aussi `B` par un lock :

```
class A {
	private B b;
	private Lock lock = new ReentrantLock();
```

```
	public A(B b) {
		this.b = b;
	}

	public Lock getLock() {
		return this.lock;
	}

	public void maMethodeCompliquée() {
		try {
			lock.lock();
			try {
				this.b.getLock().lock();
				// code compliqué ici
			} finally {
				this.b.getLock().unlock();
			}
		} finally {
			lock.unlock();
		}
	}
}

class B {
	private A a;
	private Lock lock = new ReentrantLock();

	public B(A a) {
		this.a = a;
	}

	public Lock getLock() {
		return this.lock;
	}

	public void maMethodeCompliquée() {
		A autre = null;
		do {
			try {
				lock.lock();
				if (this.a == null)
					throw new IllegalStateException();
				autre = a;
			} finally {
				lock.unlock();
			}
			try {
				autre.getLock().lock();
				lock.lock();
				if (autre == this.a) {
					// code compliqué ici
					return;
				}
			} finally {
				lock.unlock();
				autre.getLock().unlock();
			}
		} while (true);
	}
}
```

Grâce aux locks, il est possible de synchroniser sur tous les éléments d'une collection :

```
	public synchronized double getTotalDeTout(){
		double total = 0;
		// il faut prendre les locks toujours dans le même ordre
		// on va donc trier les commandes
		List<Commande> commandesTriees =
			new ArrayList<Commande>(commandes);
		Collections.sort(commandesTriees, new Comparator<Commande>() {
			@Override
			public int compare(Commande arg0, Commande arg1) {
				return System.identityHashCode(arg0)-
					System.identityHashCode(arg1);
			}
		});
		try {
			// acquisition de tous les locks
			for (Commande c: commandesTriees) {
				c.getLock().lock();
			}
			// calcul du total
			for (Commande c: commandes) {
				total += c.getMontant();
			}
		} finally {
			// libération de tous les locks
			for (Commande c: commandesTriees) {
				c.getLock().unlock();
			}
		}
		return total;
	}
```