heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Ecole d'Ingénieurs de l'état de Vaud
Eric Lefrançois
1er Février 2018

Annexe

Complément pour le mini-projet Client-Serveur

Threads en Java

Contenu

1 Programmation concurrente en Java 3
- Notion de Thread 3
- Client-Serveur - Pourquoi utiliser plusieurs threads dans la même application ? 3
- Un premier exemple : le lièvre et la tortue 4
- Concurrence ou parallélisme ? 5

2 Diagramme d'états et de transitions 6

3 Réaliser des objets actifs en Java 9
- Méthode 1- `extends Thread` 9
- Méthode 2 - `implements Runnable` 10
- Méthode 3 – Une variante, avec classe interne anonyme 10

4 Scheduling : politique d'allocation du processeur 12
- Le modèle préemptif de Java 12
- La priorité des threads 12
- Le time slicing 13
- Un exemple 13
- Le risque de famine 15
- En conclusion 15

5 Les threads démons 16

6 Exclusion mutuelle : verrouillage d'un objet 17
- Notion de moniteur 17

- Le mot-clé «synchronized» 18
- Le bloc d'instructions synchronisé 18
- Les méthodes synchronisées 19
- Seuls les blocs synchronisés sont contrôlés !! 19
- Blocs synchronisés ou méthodes synchronisées ? 19
- La synchronisation de méthodes statiques (méthodes de classes) 20

7 Rendez-vous entre threads 21
- Le mécanisme offert par Java 21
- Dans le détail.. 22
- Un modèle standard pour utiliser «wait()» 24

8 Exercice Lecteurs-Redacteurs 25

1 Programmation concurrente en Java

Notion de Thread

Les « threads » permettent à une application Java de définir plusieurs unités d'exécution se déroulant en concurrence à l'intérieur du même processus, processus correspondant au lancement de la machine virtuelle avec l'utillitaire `java`.

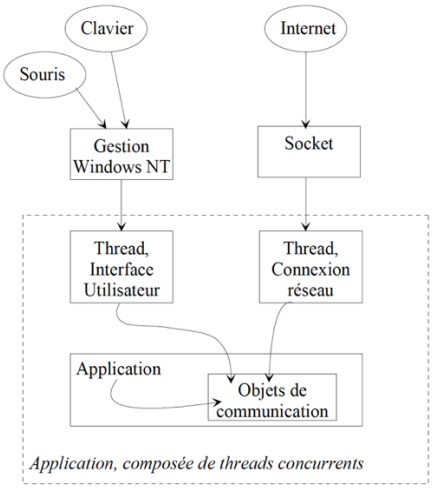
Les threads sont des « processus légers » qui possèdent les propriétés suivantes :

- Ils disposent d'un espace de mémoire partagé
  Concrètement, en Java, cela signifie que deux threads peuvent communiquer de l'information au travers d'un objet qui leur sera associé, ou au travers de variables d'instance.
- Deux threads peuvent partager les mêmes instructions (« partage de code »)
  Concrètement, en Java, cela signifie que deux threads peuvent invoquer la même méthode.
- Deux threads peuvent partager les mêmes ressources.
  Par exemple, deux threads peuvent accéder simultanément au même fichier qui n'aura été ouvert qu'une seule fois.
- Les threads ne sont pas protégés les uns des autres : si l'un d'eux corrompt l'application, ou simplement plante, c'est toute l'application (et tous les autres threads) qui plante.

Client-Serveur - Pourquoi utiliser plusieurs threads dans la même application ?

Dès qu'une application doit gérer des événements provenant de plusieurs sources différentes, et que la gestion de chacun de ces événements doit être opérée de manière indépendante et concurrente, le programmeur aura tout intérêt à confier le traitement de chacune de ces sources d'événements à un thread particulier.

Typiquement, dans le cadre d'une application Client-Serveur (mini-projet GEN), l'application sera confrontée à deux sources d'événements : des événements de l'interface utilisateur et des événements réseau. Les deux threads correspondant communiqueront entre eux, et avec l'application, au moyen d'objets déclarés à l'intérieur de l'application.



Application, composée de threads concurrents

Un premier exemple : le lièvre et la tortue

En Java, un thread est implanté sous la forme d'un objet qui exécute les instructions d'une méthode particulière : la méthode `run()`.

En tant qu'objet, un thread est susceptible de recevoir des messages : **`start()`**, **`stop()`**, **`suspend()`** etc…

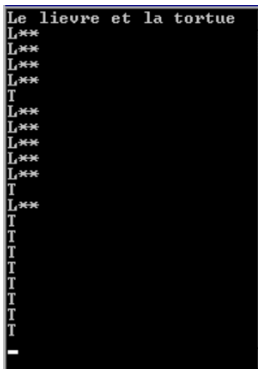
Pour créer de tels objets, Java propose la classe **`Thread`** qui met en œuvre toute la mécanique de gestion des activités. En outre, cette même classe définit une méthode **`run()`** qui, par défaut, ne fait rien.

Le programme présenté ci-dessous met en jeux un lièvre et une tortue, deux objets de type `Thread`. Ces deux objets sont instanciés à partir de la classe `Animal`, classe dérivée de `Thread`, et dans laquelle se trouve redéfinie la méthode `run()` qui sera exécutée simultanément par le lièvre et la tortue.

Le lièvre affiche des « L » toutes les 20 msec et la tortue des « T » toutes les 100 msec.

Lequel arrive en premier ?

```
class Animal extends Thread {
    // Deux variables d'instance
    private String nom;      // Nom de l'animal
    private int période;     // Période entre deux affichages

    public Animal (String nom, int période) {
        this.période = période;  this.nom = nom;
    }

    public void run() {              // Code de l'activité
        int distance = 10;           // Distance à parcourir
        while (distance > 0) {
            try {Thread.sleep(période);}    // Pause: l'animal court..
            catch(InterruptedException e) {}
            System.out.println (nom);
            distance--;
        }
    }
}

public class LievreEtTortue {
    public static void main (String arg[]) {
        System.out.println ("Le lievre et la tortue");

        Animal lièvre = new Animal ("L**", 20);
        Animal tortue = new Animal ("T", 100);
        tortue.start();       // Démarrer les activités
        lièvre.start();

        try {int j=0; j = System.in.read();}    // Attendre RETURN
        catch (IOException e){};
    }
}
```

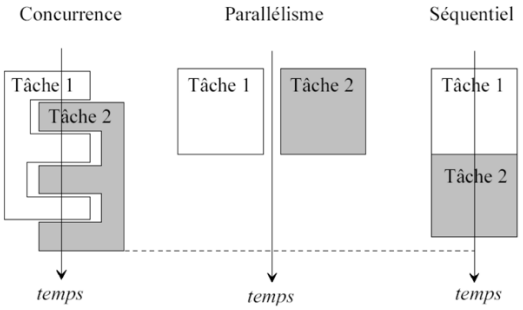


Dans cet exemple, les deux threads se partagent la ressources « écran d'affichage ». Ils se partagent aussi le code de la classe « Animal ». Ce partage aurait difficile à opérer en utilisant des processus (les 2 fenêtres auraient été séparées, duplication du code), sans parler de la perte d'efficacité au niveau de la rapidité d'exécution.

Concurrence ou parallélisme ?

Les systèmes concurrents donnent simplement l'illusion d'avoir plusieurs activités qui s'exécutent en même temps. En fait, toutes ces activités se partagent l'unique processeur de la machine par un entrelacement de leur exécution.

Les systèmes parallèles sont capables d'exécuter simultanément plusieurs activités. A cette fin, ces systèmes disposent de plusieurs processeurs, responsables chacun de l'exécution d'une activité.



Que fait Java

Les concepteurs de Java ont décidé d'exploiter le parallélisme offert par certaines plateformes. C'était le cas par exemple – *à l'époque de la conception de Java* – du système d'exploitation Windows NT qui était capable d'exécuter plusieurs threads de manière simultanée sur plusieurs processeurs.

Ainsi, plutôt que d'implémenter la concurrence des threads entièrement dans la machine virtuelle, les concepteurs ont choisi de s'appuyer sur la mécanique sous-jacente mise à disposition par le système d'exploitation.

Il est donc possible que plusieurs threads de votre application s'exécutent en vrai parallélisme !

Cette façon d'avoir envisagé les choses présente un gros inconvénient : à moins que l'on y prenne garde, **les applications Java ne sont plus portables d'un système à l'autre** !

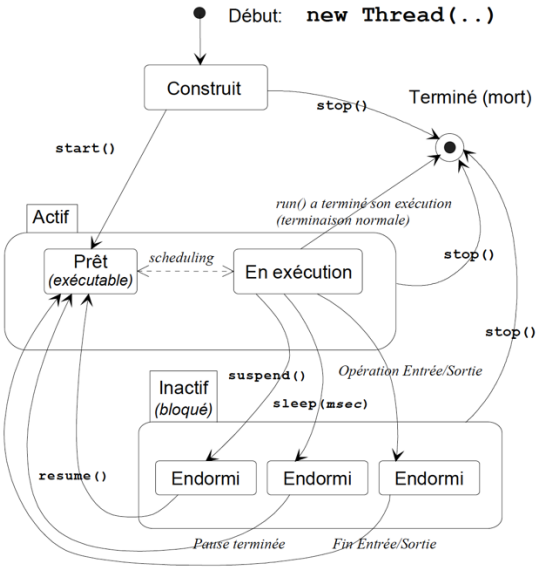
Certaines caractéristiques, – comme le « time-slicing » notamment –, n'apparaîtront que sur certains systèmes.

En dehors de ça, Java propose une implémentation des threads qui essaye, autant que possible, d'être indépendante du système d'exploitation. Ainsi, la politique d'allocation du processeur sera définie par un certain nombre de principes ou de règles qui devront être observées par les différentes implantations de la machine virtuelle.

2 Diagramme d'états et de transitions

Comme nous l'avons vu précédemment, un thread est implanté sous la forme d'un objet, susceptible de répondre à différents messages : `start()`, `stop()`, etc…

Un diagramme d'états et de transitions nous donnera une première idée de la logique de mise en œuvre des threads en Java.



- **Construction d'un thread**

  Passage à l'état Construit.

  L'objet vient d'être créé par « `new` », mais le thread est « dans les limbes » : il n'est pas encore exécutable. Il faudra pour cela lui envoyer le message `start()`.

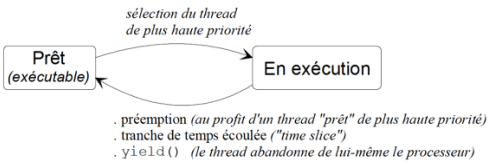
- **Lancement d'un thread : `start()`**

  Passage à l'état Actif.

  La méthode `run()` a été lancée (le message `start()` a été reçu).

  Le thread fait alors partie des threads qui ont le droit de s'exécuter : le thread alterne entre les états Prêt et En exécution, en fonction de la politique d'ordonnancement des threads.

  C'est le « scheduler » (l'ordonnanceur) qui peut faire revenir le thread de l'état En exécution à l'état Prêt.



- **Blocage d'un thread** (**`sleep()`**, **`suspend()`**)

Passage à l'état Inactif .

On notera qu'à chaque cas de blocage (d'endormissement) correspond une et une solution pour le réveil (passage à l'état Actif).

`sleep(..)`	⇨	Fin de la pause
`suspend()`	⇨	Réception du message `resume()`
Entrée/Sortie	⇨	Opération terminée (p.e : `System.in.read() ;`)

Voici un exemple de suspension d'un thread: la bavarde Hildegarde

```
class Bavarde extends Thread{
    public void run () {
        while (true) {
            afficher ("Blabla") ;
        }
    }
}

Bavarde Hildegarde = new Bavarde() ;
Hildegarde.start () ;

afficher ("Hildegarde, " ) ;
afficher ("nous avons ta mere et moi, " ) ;
afficher ("Qqch a te dire ") ;
afficher ("Hildegarde veux-tu bien m'ecouter ? ") ;

Hildegarde.suspend() ;
afficher ("nous avons decide de te marier " ) ;
afficher ("J'espere que tu es contente ") ;
Hildegarde.resume() ;
```

Dans la mesure où la méthode `afficher(..)` suspend la tâche courante le temps que l'opération d'entrée-sortie soit terminée, nous aurons l'affichage suivant:

```
Hildegarde,
BlaBla
nous avons ta mere et moi,
BlaBla
Qqch a te dire
BlaBla
Hildegarde veux-tu bien m'ecouter ?
BlaBla
nous avons decide de te marier
J'espere que tu es contente
BlaBla
BlaBla
…
```

- **Mort d'un thread**

  La mort d'un thread survient après deux occasions:

  - Fin de la méthode **`run()`**

    ⇨ le thread se suicide …

  - Réception du message **`stop()`**

    ⇨ le thread est assassiné (solution simple et radicale, mais peu recommandée, en tous les cas en programmation)

Supposons maintenant que le père soit vraiment fatigué des bavardages de sa fille, c'est très simple :

```
Hildegarde.stop();
System.out.println (" Enfin la paix ! ");
```

- **Récupération d'un thread « mort » par le garbage collector**

  Pour être récupéré par le garbage collector, le fait que le thread ait terminé son exécution ne suffit pas. Il faut encore que l'objet qui le représente ait été déréférencé.

  Le cas échéant, il suffit d'opérer cette opération manuellement : `activité = null ;`

  Le système de gestion des threads s'occupera alors lui-même de libérer toutes les ressources allouées au thread.

  Au cas où l'objet représentant le thread est déférencé sans que le thread ne soit mort, que l'on se rassure: le thread ne sera pas détruit pour autant. Le système de gestion des threads conserve en effet une référence sur tous les threads actifs.

- Contrôler l'état d'un autre thread **`isAlive()`**

  La méthode `isAlive()` retourne `true` dans tous les cas où le thread est dans l'un ou l'autre des deux états Actif ou Inactif.

- Attendre la terminaison d'un thread **`join()`**

  Le message `join()` envoyé à un thread «`t`», bloque le thread courant en attendant la terminaison du thread «`t`».

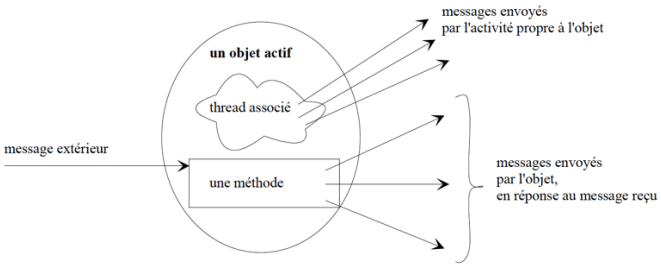
  Le thread courant est réveillé aussitôt que le thread «`t`» passe à l'état Terminé (mort).

3 Réaliser des objets actifs en Java

Vis à vis de la concurrence, la méthodologie Objet (selon [Booch]) parle des **objets actifs**.

Un objet actif est associé à un thread :

- Qui le rendra capable de changer d'état par lui-même,
- Qui le rendra capable de générer par lui-même des messages vers d'autres objets.



Comment réaliser un « objet actif » en Java ?

En gros, Java prévoit deux manières d'opérer.

- **Méthode no1 :** L'objet actif « **est-un** » thread, c'est-à-dire une instance d'une sous-classe de `Thread`.
- **Méthode no2** (la plus générale, la meilleure ..)
  L'objet actif est associé à un thread, qui sera généralement déclaré sous la forme d'une variable d'instance de l'objet actif.

  Cette méthode est meilleure pour deux raisons :
  - Elle est plus naturelle du point de vue de la méthodologie objet,
  - L'objet actif peut hériter d'autre chose que de la classe «Thread» (rappelons que Java ne connaît pas l'héritage multiple).

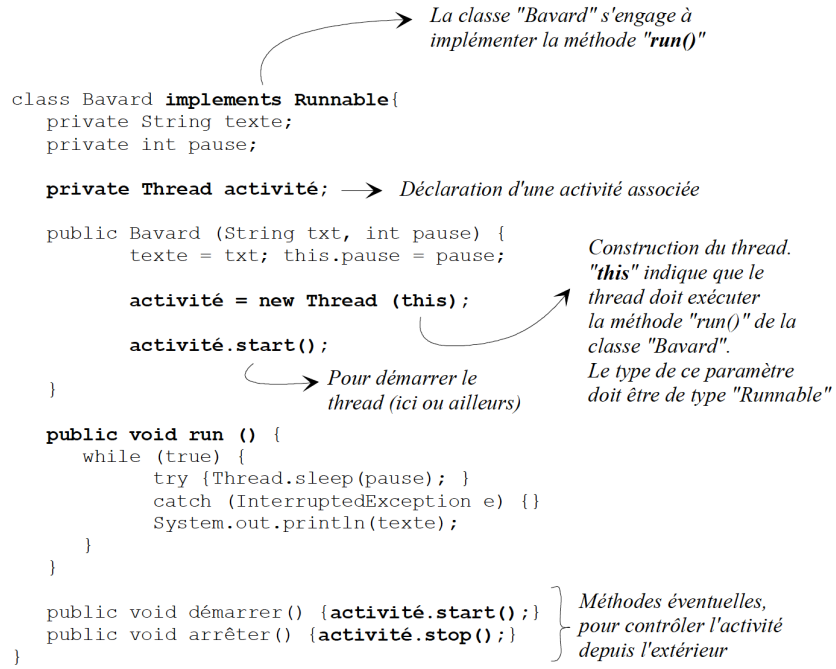
A titre d'exemple, nous allons déclarer des bavards qui passeront leur temps à afficher un texte (toujours le même) à intervalles de temps réguliers.

Méthode 1- `extends Thread`

```
class Bavard extends Thread{
    private String texte;
    private int pause;

    public Bavard (String txt, int pause) {
        texte = txt; this.pause = pause;
    }
    public void run () {
        while (true) {
            try {Thread.sleep(pause); }
            catch (InterruptedException e) {}
            System.out.println(texte);
        }
```

```
        }
    }


    public class Bavards {

        public static void main (String arg[]) {

            new Bavard ("ttt", 30).start();
            new Bavard ("xxxxx", 100).start();
        }
    }
```

Méthode 2 - `implements Runnable`



L'interface **`java.lang.Runnable`** est défini comme suit:

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

Pour essayer ce « bavard » :

```
    Bavard unBavard = new Bavard ("tttt", 30);
    unBavard.démarrer();
```

Méthode 3 – Une variante, avec classe interne anonyme

```
class Bavard {
    private String texte;
    private int pause;
    private boolean arreter = false;

    public Bavard (String txt, int pause) {
        texte = txt; this.pause = pause;
```

```
    }

    public void demarrer() {
        new Thread(){
            public void run () {
                while (!arreter) {
                    try {Thread.sleep(pause); } catch (InterruptedException e) {}
                    System.out.println(texte);
                }
            }
        }.start();
    }

    public void arreter() {arreter = true;}
}
```

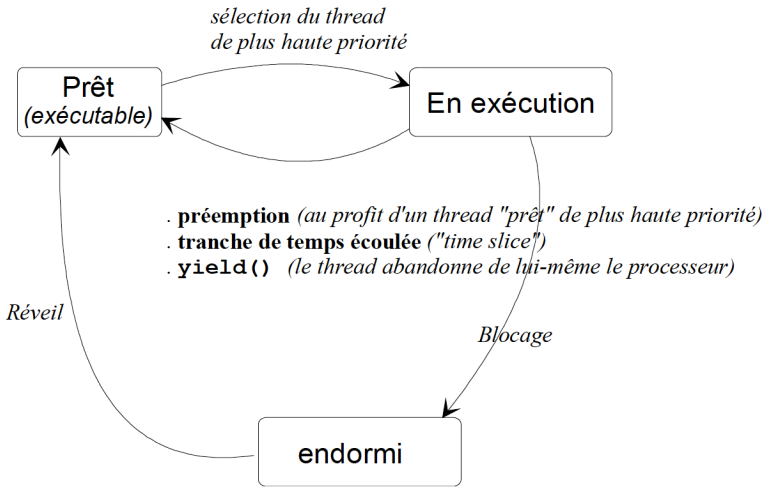
4 Scheduling : politique d'allocation du processeur

Java doit pouvoir s'exécuter à la fois sur des machines monoprocesseurs et multiprocesseurs, que le système d'exploitation soit ou non « multi-threaded ». C'est pourquoi les règles relatives aux threads sont générales.

Le modèle préemptif de Java

Le modèle d'ordonnancement de Java est dit « préemptif ».

L'exécution du thread courant peut être interrompue à tout moment au profit d'un thread se trouvant dans l'état **Prêt**.



Reprenons les différents éléments présentés dans la première partie du document. La préemption peut arriver à 4 occasions :

1. Quand le thread courant se bloque : entrée-sortie, `sleep()`, `suspend()`, ..
2. Quand le thread courant « donne la main » (`yield()`) ;
3. Quand un thread de plus haute priorité arrive à l'état **Prêt** ;
4. A l'expiration d'une tranche de temps (time-slicing)

Le point 3) et le point 4) , dépendant du système d'exploitation, introduisent une forme de non-déterminisme dans l'ordre d'exécution des threads. Si le programmeur désire que ses threads s'exécutent selon un ordonnancement précis, il doit s'en assurer en utilisant les mécanisme de synchronisation (`wait`, `notify`, ..)

La priorité des threads

Java attache une priorité à chaque thread. La priorité d'un thread est initialement la même que la priorité du thread qui l'a créé.

La priorité standard pour un thread vaut par défaut **`Thread.NORM_PRIORITY`**.

Cette priorité peut être changée en utilisant la méthode **`setPriority`** avec une valeur comprise entre les constantes **`Thread.MIN_PRIORITY`** et **`Thread.MAX_PRIORITY`**.

Ces constantes valent respectivement **`5`** (valeur par défaut), **`1`** (min) et **`10`** (max).

Dans le cas particulier des applets, la priorité maximum est limitée à NORM_PRIORITY+1, c'est-à-dire à «6».

Note d'implémentation

La mise en œuvre des priorités s'appuie sur le système d'exploitation sous-jacent.

- La plupart du temps, les systèmes ignorent purement et simplement les priorités ! 
- D'autres en tiennent compte mais uniquement dans une certaine mesure, en vous garantissant par exemple que les 3 priorités `NORM_PRIORITY`, `MIN_PRIORITY` et `MAX_PRIORITY` sont distinguées mais pas les autres..

Il est donc fortement conseillé, si on désire être portable, de ne pas construire son système d'ordonnancement sur la base des priorités.

Le time slicing

Là encore, le comportement dépend du système d'exploitation sous-jacent.

- **Windows** : Ce dernier organisera les threads à l'état Prêt en plusieurs queues, une par priorité (priorité gérée par Windows)). Au sein d'une queue de même priorité, les threads seront organisés selon un schéma FIFO avec time slice.
- **SUN-Solaris** : Le système d'exploitation de l'époque pour les machines SUN (à l'origine de Java) : Aucun timeslice prévu !
- **Linux** : La mise en œuvre d'un time slice pour Java doit normalement être configurée

La solution au problème de portabilité ?

⇨ Utiliser l'opération «yield()» : Demander à un thread d'abandonner le processeur au profit d'un thread de même priorité.

Un exemple

Voici par exemple 3 « bavards » qui affichent chacun 10 fois une lettre de l'alphabet qui leur est propre :

```
class Bavard extends Thread {
    private String monSigle;

    public Bavard (int priorité, String monSigle) {
        setPriority (priorité) ;
        this.monSigle = monSigle;
    }

    public void run () {
        for (int cpt = 1 ; cpt <= 3 ; cpt++) {
            System.out.println (monSigle + "  " ) ;
            yield() ;
        }
    }
}

public class LesBavards {
    public static void main (String arg[]) {
        new Bavard (5, "55555").start();
        new Bavard (1, "11111").start();
        for (int cpt = 1 ; cpt <= 3 ; cpt++) {
            System.out.println ("Thread par défaut") ;
        }
    }
}
```

Cet exemple comporte 3 threads : le « thread par défaut », qui correspond à l'exécution de «main» (priorité 5) et deux bavards, de priorité respective 3 et 5.

L'exécution du programme sous Windows est très aléatoire. La présence ou non du `yield` ne change rien au résultat (ce qui semble prouver que le `System.out.println` libère le processeur (n'est pas bloquant).

```
Thread par defaut
55555
Thread par defaut
55555
Thread par defaut
55555
11111
11111
11111
```

Ou ceci (le résultat est différent à chaque exécution)

```
Thread par defaut
55555
11111
Thread par defaut
Thread par defaut
11111
55555
11111
55555
```

Exécution du programme (sous MacOS), sans le `yield()` :

```
Thread par défaut
Thread par défaut
Thread par défaut
11111
11111
11111
55555
55555
55555
```

Ou ceci (le résultat est différent à chaque exécution)

```
Thread par défaut
Thread par défaut
```

```
Thread par défaut
55555
55555
55555
11111
11111
11111
```

→ Ne tient pas compte de la priorité.. & le `System.out.println` n'est pas bloquant (ne relâche pas le processeur)
→ Un `yield` sera donc nécessaire pour passer le contrôle entre les threads.

Avec le yield() :

```
Thread par défaut
Thread par défaut
Thread par défaut
11111
55555
11111
55555
11111
55555
```

Ou

```
Thread par défaut
Thread par défaut
Thread par défaut
55555
11111
55555
11111
55555
11111
```

Le risque de famine

Pour compliquer le tout, Java sous Windows trouve judicieux de garantir qu'un jour ou l'autre, un thread de basse priorité finira par s'exécuter, de manière à éviter le risque de famine (« starvation »).

Cela signifie qu'au sein de la queue des processus « Prêt », ça n'est de toute façon pas forcément le thread de plus haute priorité a qui le processeur est confié…

En conclusion

Pour écrire une application Java portable, il nous faut prendre en compte certaines considérations.

Comme nous ne pouvons faire aucune supposition quant au système d'exploitation sous-jacent, le programme doit être conçu pour fonctionner dans le pire des cas..

Notamment, il faut s'attendre à ce que nos threads soient interrompus à tout moment (si le système implémente un time-slice). Il nous faut donc utiliser le verrou de synchronisation mutuelle (`synchronized`).

Il faut s'attendre également au contraire, à ce que nos threads ne soient jamais interrompus. Il nous faut donc utiliser des méthodes comme `sleep(..)` ou `yield()` de manière à abandonner le processeur au profit des autres.

On ne peut pas non s'appuyer sur les niveaux de priorité.

5 Les threads démons

Ce sont typiquement des threads de faible priorité qui tournent en boucle infinie, en attendant qu'on leur demande de rendre un service : nettoyer la mémoire, afficher des images, etc...

- La méthode **`setDaemon()`** permet de spécifier qu'un thread est un démon. Elle doit être utilisée avant que le thread soit démarré (`start()`) ;
- La méthode **`isDaemon()`** permet de savoir si un thread est un démon..

Quelle est la différence entre un « thread démon » et un « thread normal » (encore appelé « thread utilisateur ») ?

Aucune, sinon qu'au niveau système, la machine virtuelle Java n'attendra pas la mort d'un thread démon pour arrêter le programme : java décidera de tuer lui-même les threads démons quand il ne restera plus qu'eux en jeu.

Les threads utilisateurs, tant qu'il en reste au moins un, empêchent la machine virtuelle d'arrêter le programme.

Le garbage collector est un exemple typique de « thread démon ».

6 Exclusion mutuelle : verrouillage d'un objet

Dans cette partie du document, nous allons étudier comment éviter les problèmes liés au « hasard de l'ordonnancement », comment coordonner l'accès à l'information, et enfin comment synchroniser les threads de manière générale.

Notion de moniteur

Nous verrons que la synchronisation en Java est basée sur la technique des « moniteurs » de C.A.R Hoare. Un moniteur est un bout de code dont l'accès exclusif est garanti par un «verrou d'exclusion mutuelle » : un seul thread à la fois sera en possession du verrou, ce qui lui donnera la possibilité d'exécuter les instructions du moniteur.

Nous allons illustrer le problème avec l'exemple classique qui suit.

Supposons que l'on dispose d'un distributeur de billets de banque qui permette de faire des retraits sur un compte bancaire particulier.

L'algorithme réalisé par le distributeur est le suivant :

1. vérifier que le solde du compte est suffisant pour autoriser le retrait,
2. mettre à jour l'état du compte (nouveau solde),
3. distribuer les billets,
4. imprimer un reçu

Voici le code de cet algorithme en Java :

```
public class Distributeur {

    public void retirer (int montant) {
        CompteBancaire cb = getCompte() ;
        if (cb.débiter(montant)) {
            distribuer(montant) ;
            imprimerReçu() ;
        }
    }

   :
}

public class CompteBancaire {
    private int solde ;
    public boolean débiter (int montant) {
        if (solde – montant >= 0) {
            solde –= montant ;
            return true ;
        }
        return false ;
    }
}
```

En règle générale, cet algorithme fonctionnera très bien jusqu'au jour ou deux personnes voudront opérer un retrait sur le même compte simultanément. On peut supposer par exemple qu'il s'agit d'un « compte-joint » partagé par un couple.

Avec le « hasard de l'ordonnancement », il est possible (malgré le contrôle) que l'argent soit distribué aux deux parties, et que l'on se retrouve avec un solde négatif !

Supposons en effet que les opérations soient accomplies dans cet ordre :

1. Le thread du mari commence l'exécution de la méthode «débiter» , et il est vérifié que le solde est supérieur ou égal au montant à débiter ;

```
public boolean débiter (int montant) {
    if (solde – montant >= 0) {

                          *----------------------------------> préemption

            solde -= montant ;

            return true ;

      }

        return false ;
}
```

2. **préemption** : le thread de la femme prend le contrôle du processeur et commence l'exécution de la méthode «débiter »; comme le montant du mari n'a pas encore été débité, le retrait est accepté, la méthode retourne «`true`» et l'argent est distribué ;
3. Le thread du mari reprend le contrôle du processeur : la méthode `débiter` continue son exécution (entre temps, la valeur du solde a été modifiée) et l'argent est distribué, même si le solde est passé en négatif !

Une solution simple pour ce problème consiste à « **atomiser** » la méthode `débiter`.

Méthode atomique

Une méthode atomique ne peut pas être exécutée par deux threads simultanément. Pratiquement, une telle méthode n'est pas « décomposable » : elle ne peut pas être interrompue en son milieu pour être exécutée par un autre thread.

Section critique

En utilisant une autre terminologie, on dira que le code d'une méthode atomique s'inscrit dans une « **section critique** », c'est-à-dire une section qui ne peut être exécutée que par un seul thread à la fois.

Le mot-clé «synchronized»

En Java, la mise en œuvre d'une méthode atomique s'opère très simplement au moyen du mot-clé **`synchronized`**. A lui seul, il garantit que cette méthode ne sera exécutée que par un seul thread à la fois.

```
public synchronized boolean débiter (int montant) {
        ...
}
```

Comment ça fonctionne ? Par la manipulation implicite d'un « verrou d'exclusion mutuelle »…

Le bloc d'instructions synchronisé

Chaque objet est associé à un « **moniteur** ». Ce dernier contrôle l'accès au code de l'objet au moyen d'un « **verrou** ».

Le verrou peut être ouvert ou fermé.

Le verrouillage d'un objet s'opère par le biais d'un **bloc de synchronisation :**

```
synchronized(unObjet) {
        instructions
```

```
}
```

Le thread qui exécute un tel bloc :

- Demande à verrouiller l'objet `unObjet`
- Si l'objet n'est pas déjà verrouillé
  1. Obtention du verrou par le thread
  2. Le thread peut alors exécuter les instructions du bloc
  3. A la fin du bloc d'instructions, le verrou est libéré automatiquement

- Si l'objet est déjà verrouillé
  1. Le thread est bloqué (perd le processeur). Il est inscrit dans la liste d'attente associée au verrou.
  2. Le thread passera à l'état Prêt une fois que le verrou sera libéré et que son tour sera venu. A la libération d'un verrou, un des threads se trouvant dans la liste d'attente **est choisi arbitrairement** pour acquérir le jeton.
  3. Pour exécuter le bloc d'instructions, le thread doit encore « attendre » son passage à l'état En exécution.

Les méthodes synchronisées

Il arrive souvent que toutes les instructions d'une méthode soient placées dans une **section critique**, comme ci-dessous :

```
public void uneMéthode (..) {
        synchronized(this) {
                instructions de la méthode
        }
}
```


Ainsi programmée, une telle méthode ne pourra être exécutées que par un thread à la fois !

Il est alors possible, pour simplifier l'écriture, de **synchroniser** la méthode elle-même, ce qui revient, – *pour finir* -, exactement au même :

```
public synchronized void uneMéthode (..) {
        instructions de la méthode
}
```

Seuls les blocs synchronisés sont contrôlés !!

Précisons pour clarifier les choses que seuls les blocs synchronisés sont contrôlés : les méthodes non synchronisées d'un objet peuvent être exécutées simultanément par plusieurs threads, même si l'objet a été « verrouillé ».

Blocs synchronisés ou méthodes synchronisées ?

De manière générale, les sections critiques doivent être les plus courtes possibles. On améliore ainsi le temps de réponse du système : pendant qu'un thread occupe une section critique, les autres threads qui aimeraient y avoir accès sont bloqués en attendant la libération du verrou. C'est ennuyeux. Par ailleurs, plus les sections critiques sont courtes, plus les chances d'interblocage sont réduites..

Comme on l'a vu, une méthode synchronisée pourrait être écrite ainsi :

```
public void uneMéthode (..) {
```

```
        synchronized(this) {
                instructions de la méthode
        }
}
```

Il faut alors se poser la question suivante : est-il vraiment nécessaire de synchroniser toutes les instructions de la méthode ? A répondre de cas en cas, suivant le contexte.

La synchronisation de méthodes statiques (méthodes de classes)

Chaque objet est associé à un verrou. C'est également le cas des classes…

Il est donc possible de déclarer qu'une méthode statique est synchronisée :

```
public synchronized static uneMéthodeDeClasse(..) {…}
```

ou encore de synchroniser un bloc d'instructions sur une classe à partir de l'une de ses variables statiques :

```
synchronized (variableStatique) {
        instructions
}
```

7 Rendez-vous entre threads

Dans le domaine de la programmation concurrente, il ne suffit pas d'éliminer les problèmes dus au hasard de l'ordonnancement (les « race conditions »). Les threads doivent aussi avoir la possibilité de coopérer.

En effet, si tous les threads de votre application sont capables de s'exécuter de manière tout à fait indépendante les uns des autres, le mécanisme d'exclusion mutuelle peut suffire.

Toutefois, dans la plupart des cas, les threads concurrents seront amenés à communiquer entre eux : échanges d'informations ou d'événements. **Il ne s'agit plus d'une compétition, mais plutôt d'une coopération**.

Le mécanisme offert par Java

Un ou plusieurs threads pourront attendre « qu'il se passe quelque chose » sur un objet. C'est un autre thread qui mettra fin à leur attente.
En d'autres termes, un thread pourra se mettre en attente d'un événement. Il attendra alors qu'un autre thread l'informe, le notifie, de l'occurrence de l'événement sur le même objet que celui sur lequel le thread s'est mis en attente.

C'est possible grâce aux méthode «**`wait`**», «**`notify`**» et «**`notifyAll`**».

Si un thread **`Ta`** envoie le message **`wait()`** à un objet quelconque **`x`**, le thread `Ta` est bloqué en attendant qu'il se « passe quelque chose » en rapport avec cet objet `x`.

Le déblocage aura lieu dès qu'un thread **`Tb`** aura signifié à l'objet «`x`» « qu'il s'est passé quelque chose » en lui envoyant le message **`notify()`**, ou **`notifyAll()`**.

Si plusieurs threads ont envoyé le message **`wait()`**, ils seront tous inscrits dans une file d'attente associée à l'objet.

- Le message **`notifyAll()`** aura pour effet de débloquer tous les threads se trouvant en file d'attendre.
- Le message **`notify()`** débloque uniquement un thread, choisi arbitrairement dans la liste d'attente.

Voici un exemple connu : le rendez-vous des étudiants avec leur notes…

```
class ListeNotes {

    synchronized void seFontAttendre () {
        try {
            wait() ;
        }
        catch (InterruptedException e) {}
    }

    synchronized void êtreDiffusees () {
        notify() ;
    }
}

class Eleve () {
     public Eleve (ListeNotes sesNotes) {
        :
        sesNotes.seFontAttendre()
        System.out.println ("J'ai eu mes notes") ;
        :
     }
}
class Prof {
    public Prof (ListeNotes lesNotes) () {
        :
```

```
            lesNotes.etreDiffusees()
            System.out.println ("VOICI vos notes") ;
            :
        }
    }
    ListeNotes aieAie = new Notes() ;
    Prof prof = new Prof (aieAie) ;
    Eleve eleve = new Eleve (aieAie) ;
```

- Remarquons que si jamais élève et professeur ne travaillent pas sur le même paquet de notes, il faudra tuer l'élève (CONTROL C) pour abréger son attente.
- Si plusieurs élèves attendent : il suffit de remplacer `notify()` par `notifyAll()`

 Notons que la méthode `wait()` dispose d'un « timeout » possible.

Dans le détail..

Regardons maintenant de manière plus formelle la sémantique des méthodes `wait()`, `notify()` et `notifyAll()`.

Du point de vue de la synchronisation, remarquons en préliminaire qu'un objet :

1. Est associé à un verrou, qui contrôle l'entrée dans le moniteur. Si le moniteur est déjà occupé, tous les threads qui désirent acquérir le verrou sont inscrits dans une liste d'attente : la liste «**`monitor_wait`**»
2. Est associé à un ensemble de threads, placés dans la liste «**`condvar_wait`**», de tous les threads qui ont envoyés le message `wait()` à cet objet.

- Sémantique de **`wait()`**

  Pour invoquer la méthode `wait()` sur un objet `x`, un thread `t` doit d'abord avoir acquis le verrou sur cet objet.

  Comme par exemple :

```
synchronized (x) {
        x.wait() ;
        :
}
```

  1. Le thread est inscrit dans la liste «**`condvar_wait`**» de l'objet a qui est envoyé le message,
  2. Le thread est bloqué,
  3. Le verrou acquis sur l'objet «x» est libéré.

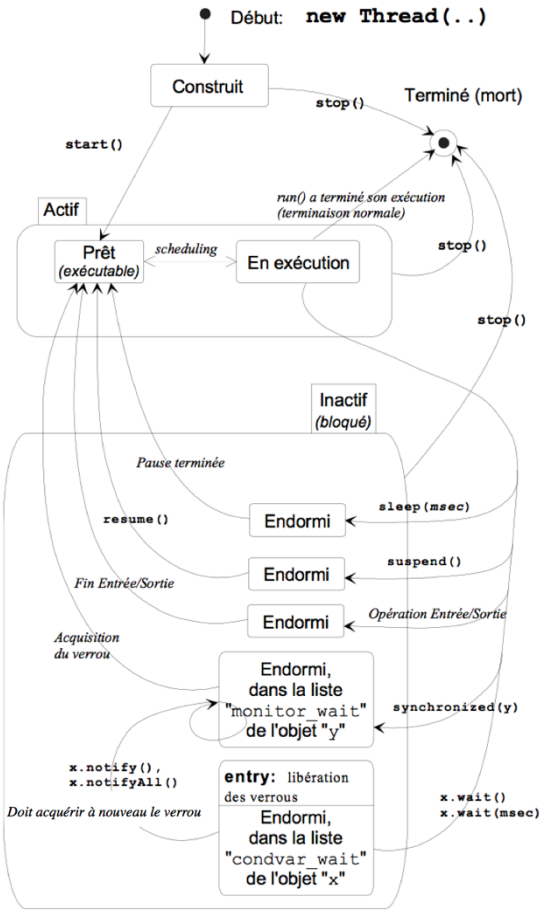
  Le réveil du thread «t» aura lieu quand :

  - Un autre thread envoie un `notify()` à l'objet `x`, et que le thread en question a la chance d'être choisi au sein de la liste `condvar_wait`. Ce choix est arbitraire, Java ne précise rien ;
  - Ou alors, quand un autre thread envoie le message `notifyAll()` à l'objet `x`. Dans ce cas, tous les threads de la liste `condvar_wait` sont réveillés ;
  - Ou enfin, quand le délai spécifié de manière optionnelle dans le message «`wait(msec)`» est échu.

  Qu'advient-il alors du thread `t` ?

  - Ce dernier est enlevé de la liste `condvar_wait` ;

- Mis en état de tentative de verrouillage de l'objet `x`. Il est mis éventuellement en compétition avec d'autres.
- Une fois entré dans le moniteur de `x`, il doit alors récupérer le verrou de l'objet qu'il avait perdu avant d'être bloqué ;
- La méthode `wait()` est alors terminée, et le thread `t` est introduit dans la liste des threads à l'état **Prêt**. Notons qu'à cet instant, le thread **t** possède toujours le verrou sur l'objet **x**.



- **Sémantique de la méthode `notify()`**

  Comme pour **`wait()`**, cette méthode peut être invoquée par un thread `t` sur un objet `x` à condition seulement que `t` ait acquis le verrou de `x` :

```
synchronized (x) {
        x.notify() ;
        :
}
```

  Un des threads de la liste **`condvar_wait`** est alors choisi au hasard afin d'être réactivé. Si cette liste est vide, la méthode `notify()` n'a aucun effet.

  Le thread `t` continue son exécution et continue notamment d'occuper le moniteur de `x`. Ainsi, le thread qui a été réactivé ne continuera son exécution qu'une fois que le thread `t` aura quitté le moniteur.

- **Sémantique `notifyAll()`**

  Fonctionnement identique à `notify()`, sinon que la liste **`condvar_wait`** est vidée complètement.

Un modèle standard pour utiliser «wait()»

Il y a un modèle standard qu'il est recommandé d'utiliser avec `wait` et `notify`. Un thread qui attend une condition pour faire quelque chose devrait appeler une méthode `faireSCondition()` qui devrait être écrite un peu comme ceci

```
class X extends Thread {

    boolean condition;   // Condition à attendre

    public void run () {
      :
        faireSiCondition();
      :
    }

    public synchronized void faireSiCondition () {

        while (!condition) {
            try {
                wait(); .. Pendant le wait, le moniteur est libéré (perte du verrou)
                    .. Après le wait on sait que  le thread se trouve à nouveau dans
                    .. le moniteur : le verrou a du être récupéré pour sortir du wait

                    .. Entre-temps..,  la condition a peut-être changé : elle n'est plus
                    .. forcément vraie.
                    .. >> Tester à nouveau la condition avec une boucle "while"
            }catch (InterruptedException e) {}
        }

    }

    public void signaler () {
           // Méthode appelée par un autre thread pour signaler que
            // la condition est arrivée
        synchronized (this) {
            condition = true;
            notify();
        }
    }
   :
   :
}
```

Au travers de cet exemple, on peut voir pourquoi les méthodes `wait()` et `notify()` doivent être utilisées en conjonction avec le verrou de synchronisation.

Ainsi le « Test-And-Set » : `if ( !condition) wait()` doit se faire de manière atomique.

Pendant le `wait`, il est obligatoire que le verrou soit libéré pour qu'un autre thread puisse envoyer son `notify()`.

Si la libération du verrou était effectuée avant le blocage, le thread risquerait d'être notifié avant même d'être bloqué. Dans ce cas, la notification serait perdue…

8 Exercice Lecteurs-Redacteurs

Compléter la classe LecteursRedacteurs écrite ci-dessous :

```
class LecteursRedacteurs {
        private Information info ;     // Info partagée par les lecteurs et les rédacteurs
        public void ecrire (BlocInfo uneInfo) { /* code non présenté */ }
        public BlocInfo lire () { /* code non présenté */ }

        private boolean unRedacteurActif ;      // Retourne vrai si un rédacteur est
                                                // actuellement actif
        private int nbRedacteursEnAttente ;     // Retourne le nombre de rédacteurs bloqués
        private int nbLecteurs ;                // Retourne le nombre actuel de lecteurs actifs
        private int nbLecteursEnAttente ;       // Retourne le nombre de lecteurs bloqués, en
                                                // en attendant que le rédacteur libère

        A compléter les méthodes suivantes :
        public void jeVeuxLire() {}             // Méthode devant être invoquée par tout
                                                // lecteur qui désire lire des infos,
                                                // le lecteur sera bloqué tant que
                                                // la ressource n'est pas libre
        public void jaiFiniDeLire()             // Méthode à invoquer par tout lecteur qui a
                                                // terminé  sa lecture
        public void jeVeuxEcrire() {}           // Méthode devant être invoquée par tout
                                                // rédacteur qui  désire écrire des infos,
                                                // ce rédacteur sera bloqué tant que la
                                                // ressource n'est pas libre
        public void jaiFiniDecrire()            // Méthode à invoquer par tout rédacteur qui a
                                                // terminé son écriture
}
```

Les règles du jeu :

- Lecture et écriture simultanées interdites
- Un seul rédacteur à la fois
- Plusieurs lecteurs silmultanés possibles
- Priorité aux rédacteurs : un lecteur n'est pas autorisé à lire si il existe au moins un rédacteur en attente. Ainsi, si un rédacteur arrive alors que d'autres sont en train de lire, il sera pris en compte tôt ou tard car plus aucun lecteur ne peut « s'inscrire »

Code du rédacteur (exemple) :	Code du lecteur (exemple) :

`lr.jeVeuxEcrire() ;`
`lr.ecrire(unBloc) ;`
`lr.jaiFiniDecrire() ;`
	`lr.jeVeuxLire() ;`
	`unBloc = lr.lire() ;`
	`lr.jaiFiniDeLire() ;`