5 Les threads démons

⇨ 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...

Deux méthodes à disposition

- `setDaemon()` → Pour spécifier qu'un thread est un démon
- `isDaemon()` → Savoir si le thread est un démon

Différence entre un thread démon et un thread "normal" ?

Aucune.. Sinon que la JVM n'attend pas la mort d'un thread démon pour arrêter le programme

⇨ Les threads démons sont automatiquement tués quand il ne reste plus qu'eux en jeu

6 *Exclusion mutuelle : verrouillage d'un objet*

Buts

- Comment éviter les problèmes dus au non-déterminisme de l'ordonnancement
- Comment coordonner l'accès à l'information
- Comment synchroniser les threads de manière générale

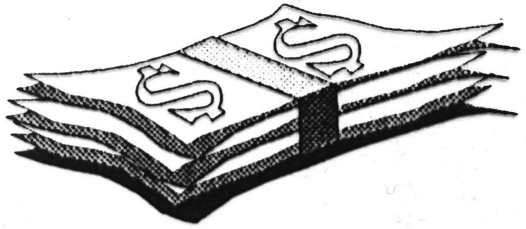
Notion de moniteur

Synchronisation basée sur la technique des **moniteurs** (C.A.R Hoare)

Exemple...

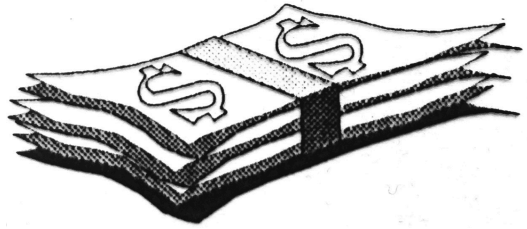
Un distributeur de billets de banque pour faire des retraits sur un compte bancaire particulier.

Algorithme du distributeur 



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

Le code de cet algorithme..



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

```
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 ;
    }
}
```

Si deux retraits simultanés..

1. Le thread du mari commence l'exécution de la méthode débiter
2. OK! Il est vérifié que le solde est supérieur ou égal au montant à débiter
3. Puis .. ***Préemption*** (le thread de la femme opère une débit)

```
public boolean débiter (int montant) {
     if (solde - montant >= 0) {
          *----------------------→ préemption
          solde -= montant ;
          return true ;
     }
     return false ;
}
```

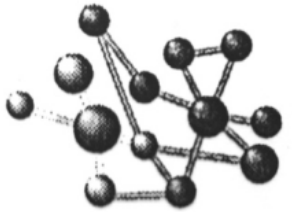
Si deux retraits simultanés..

1. Le thread du mari commence l'exécution de la méthode débiter
2. OK! Il est vérifié que le solde est supérieur ou égal au montant à débiter
3. Puis .. ***Préemption*** (le thread de la femme opère une débit)

```
public boolean débiter (int montant) {
     if (solde - montant >= 0) {
          *----------------------→ préemption
          solde -= montant ;
          return true ;
     }
     return false ;
}
```

On risque de se retrouver avec un solde négatif !!

La solution..



« **atomiser** » la méthode `débiter`

Méthode atomique

Une méthode atomique ne peut pas être exécutée par deux threads simultanément

Section critique

Autre terminologie, on dira que le code d'une méthode atomique s'inscrit dans une **section critique**

⇨ Ne peut être exécutée que par un seul thread à la fois.

Le mot-clé «synchronized»

En Java, rajouter le mot-clé **synchronized**

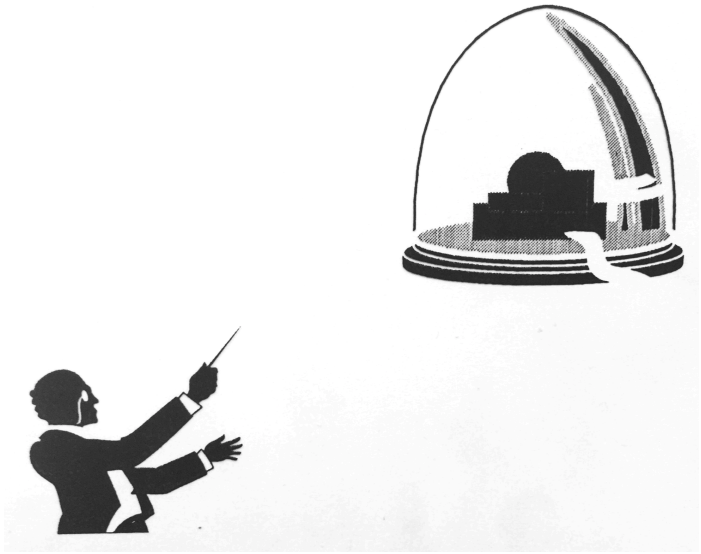
⇨ ***Méthode devient atomique***

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

Comment ça fonctionne 

⇨ Par la manipulation implicite d'un **verrou d'exclusion mutuelle**

Chaque objet est associé à un **moniteur**



- Le moniteur contrôle l'accès au code de l'objet au moyen d'un **verrou**
- Le verrou peut être ouvert ou fermé

Le bloc d'instructions synchronisé

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

- `synchronized(unObjet)` ⇨ *Demande pour verrouiller* `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 ⇨ verrou libéré

Le bloc d'instructions synchronisé

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

- **`synchronized`**`(unObjet)` ⇨ *Demande pour verrouiller* `unObjet`
- *Si l'objet est déjà verrouillé* 😟

  1. Le thread est bloqué (perd le processeur) 
  2. Il est inscrit dans la liste d'attente associée au verrou
  3. Le thread passera à l'état **Prêt** une fois que le verrou sera libéré..
     Et que son tour sera venu...

     *A la libération, un des threads de la liste d'attente est **choisi arbitrairement** pour acquérir le jeton*

Les méthodes synchronisées

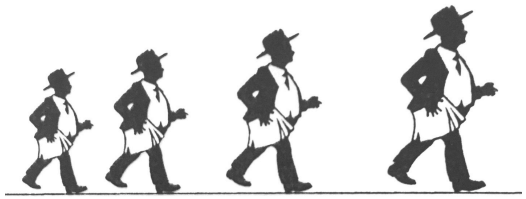
```
public void uneMéthode (..) {
   synchronized(this) {
      instructions de la méthode
   }
}
```

Equivalent à :

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

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

```
public synchronized void
uneMéthode (..) {
    instructions
}
```

OU

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

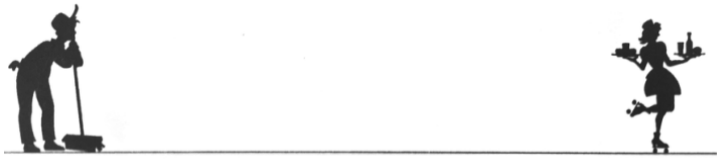
Plutôt..

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

Les sections critiques doivent être les plus courtes possibles !

- Meilleur temps de réponse du système

  ⇨ *On évite le blocage inutile des threads*



- Les chances d'interblocage sont réduites..

7 *Rendez-vous entre threads*

Exclusion mutuelle

⇨ Résoudre les problèmes de **compétition** entre les threads

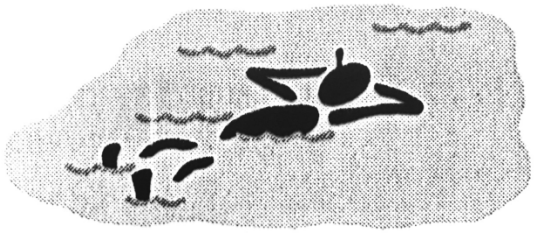
Signaler un événement – Attendre un événement

⇨ Assurer la **coopération** entre les threads

Le mécanisme offert par Java

⇨ Un ou plusieurs threads peuvent attendre *qu'il se passe quelque chose* sur un objet

```
unObjet.wait() ;
```



⇨ C'est un autre thread qui mettra fin à leur attente :

```
unObjet.notify () ;
```

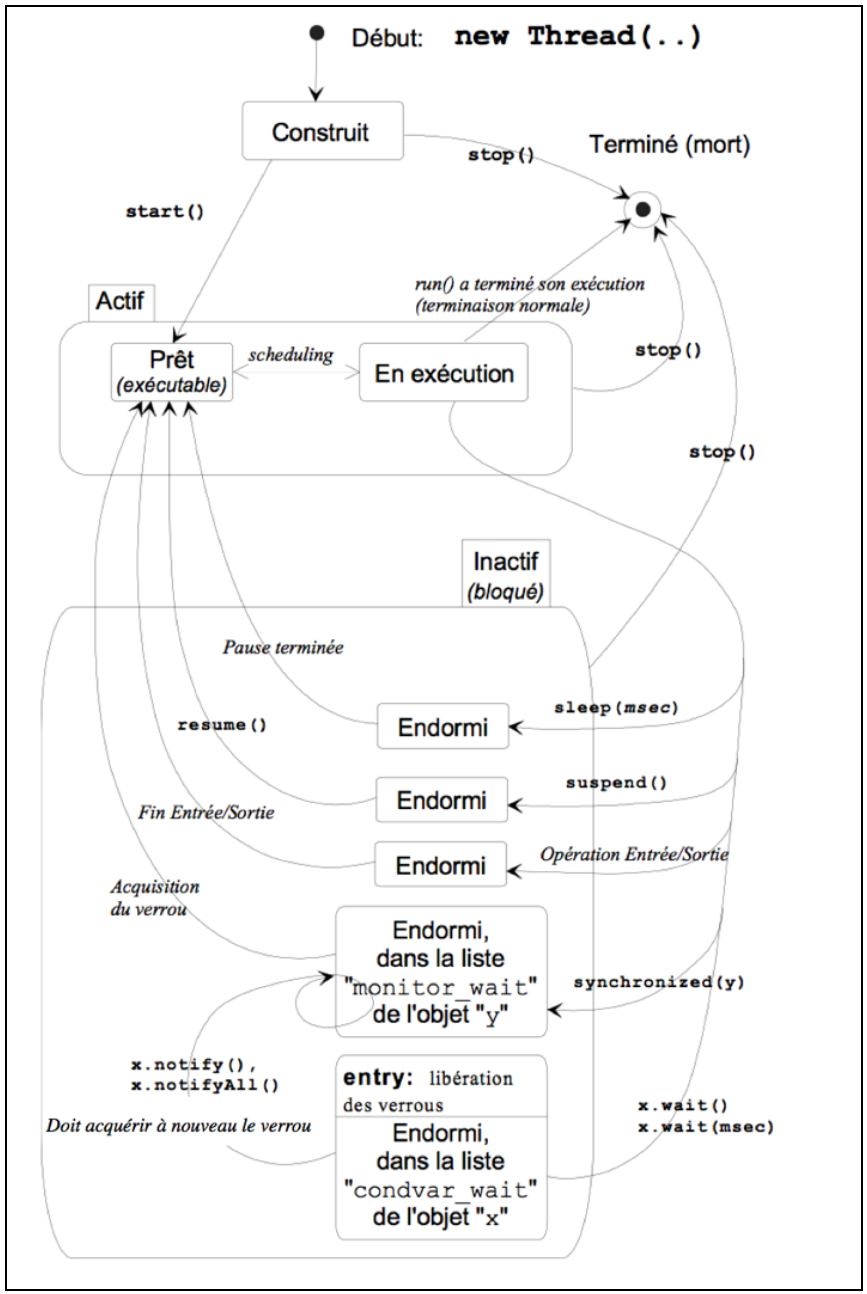
ou

```
unObjet.notifyAll () ;
```

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) ;
```

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

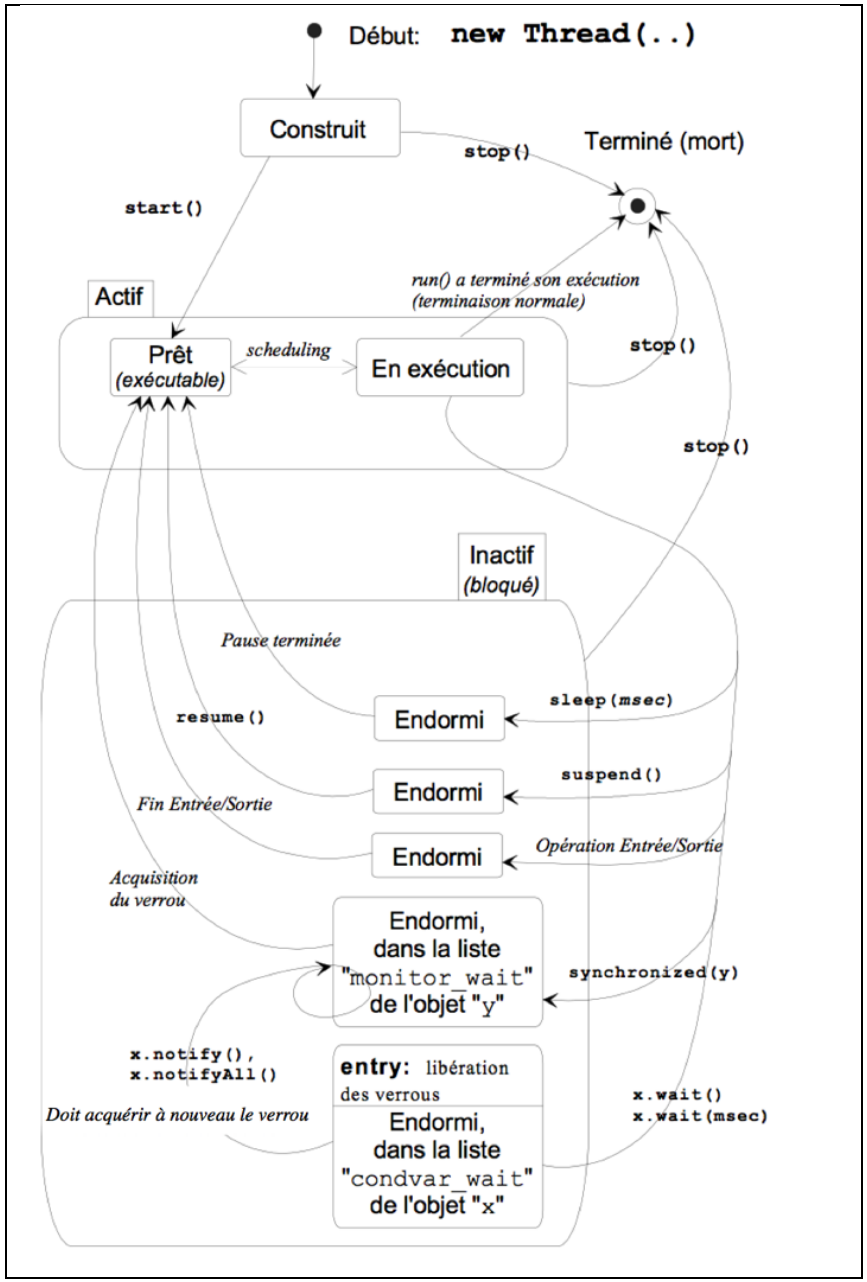


Dans le détail..

Sémantique du wait()

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

- Pour invoquer la méthode `wait()` sur un objet `x`, un thread `t` doit d'abord avoir verrouillé l'objet
- En entrant dans le `wait()`, le verrou est libéré
- Au sortir du `wait()`, le verrou doit être récupéré



Dans le détail..

Sémantique du notify()

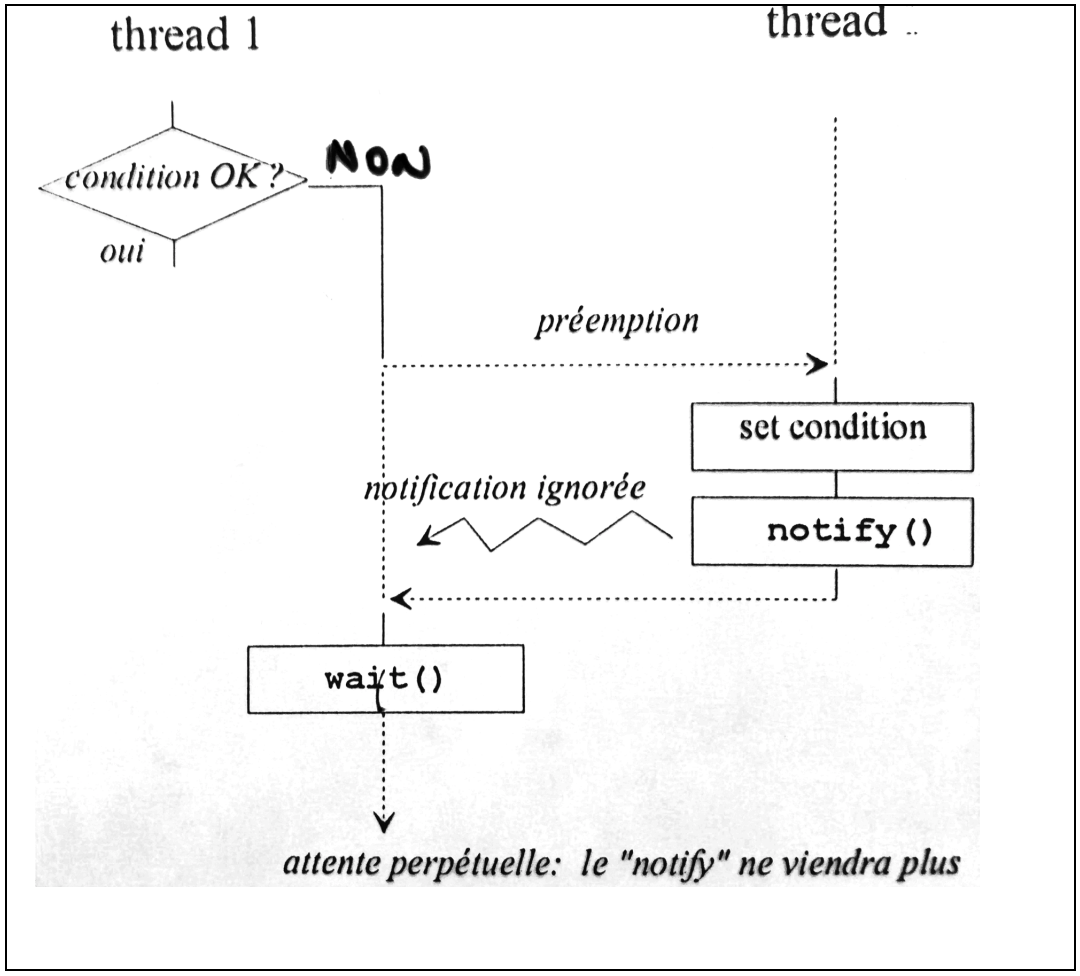
```
synchronized (x) {
    x.notify() ;
    :
}
```

- Pour invoquer la méthode `notify()` sur un objet `x`, un thread `t` doit d'abord avoir verrouillé l'objet
- Un thread de la liste `condvar_wait` est choisi **au hasard** pour être libéré
- Si cette liste est vide `notify()` n'a aucun effet
- Le thread continue normalement son exécution après `notify()`

Un modèle standard pour utiliser «wait()»

Un *faireSiCondition()* devrait être écrit 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 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();
        }
    }
    :
}
```



Le `if ( !condition) wait()`

doit se fait de manière atomique !

Sinon on risque de perdre le signal !