

Les threads

Une petite visite au zoo...

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

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

Les deux chiens aboient-ils en même temps ?

Motivation

- Dès que l'on veut que plusieurs unités de programmes travaillent en parallèle, il faut gérer la concurrence
- La concurrence repose sur:
 - * La possibilité de définir des processus concurrents:
 - en Java la technique des threads: processus léger
 - * Des mécanismes de communication entre ces processus
 - Communication synchrone par appel de méthode
 - Communication asynchrone par envoi de message
 - * Des mécanismes de synchronisation de ces processus

Processus et threads

- Dans la programmation concurrente, il existe principalement deux unités d'exécution : les **processus** et les **threads**.
- L'exécution des processus et des threads est gérée par l'OS.
- Un processus possède son propre environnement d'exécution (ressources systèmes).
- En général on a un processus par application (mais on peut faire coopérer des processus).
- La plupart des JVM tournent sur un seul processus

Processus et threads

- Un thread est souvent appelé un « processus léger» (lightweight process).
- Les threads existent dans un processus (au moins un thread par processus).
- Les threads nécessitent moins de ressources : ils partagent les ressources du processus père : mémoire, fichiers ouverts...
- L'exécution d'une JVM est «multi-threadée»: les threads constituent un aspect essentiel du langage Java.

Processus et threads

- Le multi-tâches de threads occasionne bien moins de surcharge que le multitâches de processus.
- Contrairement aux processus, les threads sont légers :
 - * ils partagent le même espace d'adressage,
 - * ils existent au sein du même processus (lourd),
 - * la communication inter-thread occasionne peu de surcharge,
 - * le passage contextuel (context switching) d'un thread à un autre est peu coûteux.
- Le multi-tâches de processus n'est pas sous le contrôle de l'environnement d'exécution java. Par contre, il y a un mécanisme interne de gestion multithreads.
- Le peu de surcharge occasionné par le système de multi-threads de Java est spécialement intéressant pour les applications distribuées. Par ex., un serveur multi-tâches (avec une tâche par client à servir).

En Java ...

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

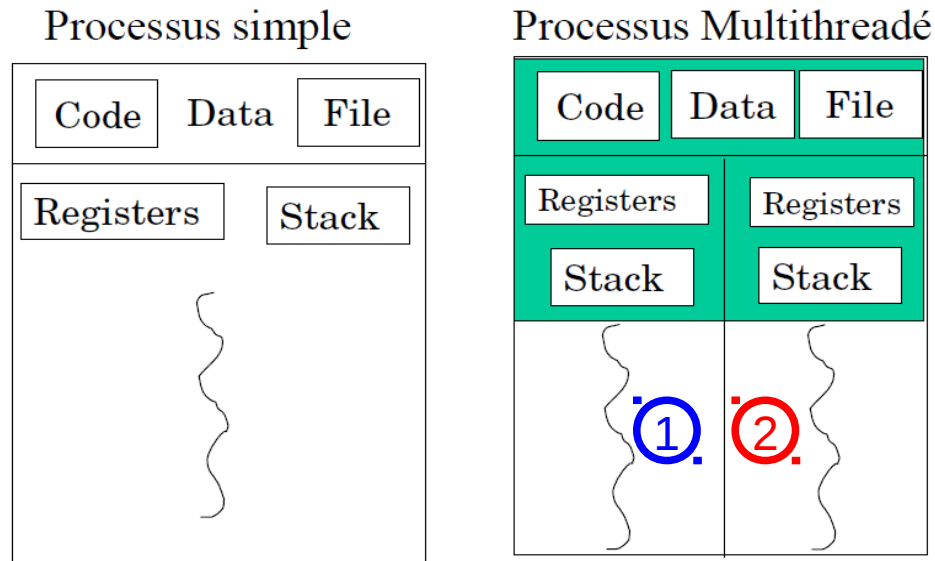
Le multithreading est intégré au langage Java. La création de threads est très simple :

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

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

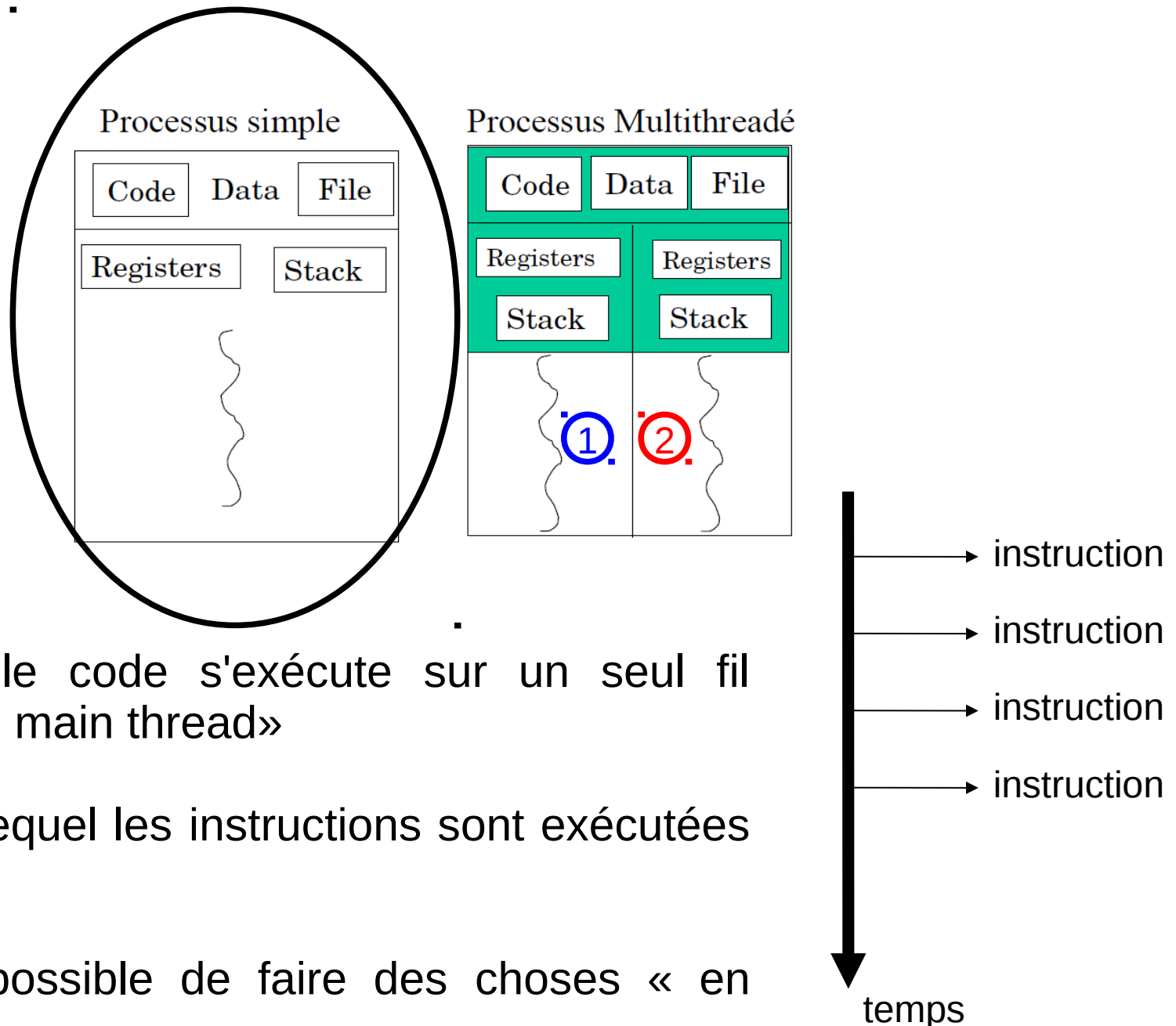


Exécution



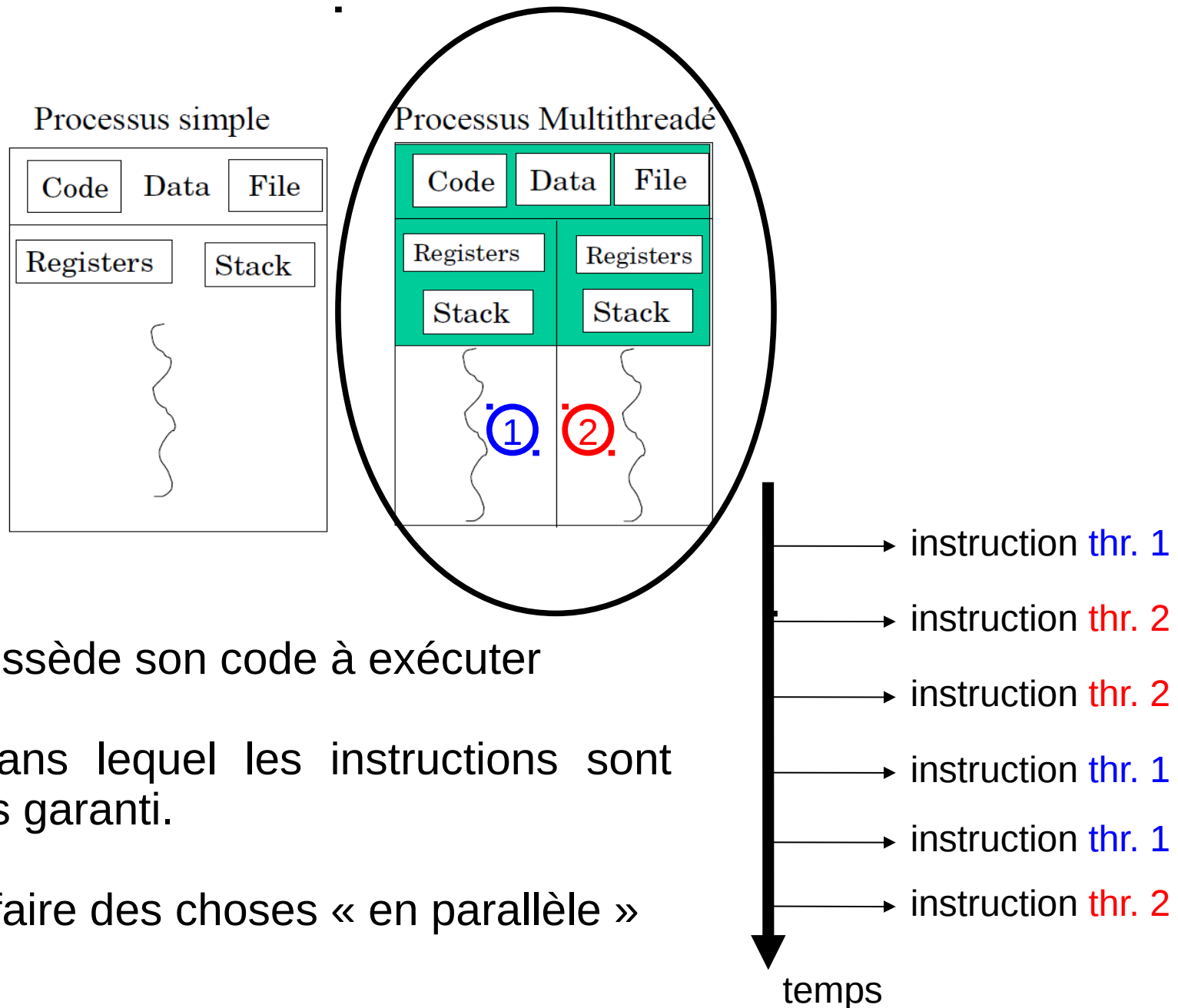
- Toute méthode d'un objet java s'exécute dans au moins un thread
- Un programme « simple » exécute les threads suivants :
 - * Thread principal
 - * Garbage collector
 - * La file d'événements graphiques

Exécution



- En général, le code s'exécute sur un seul fil d'exécution: le « main thread »
- L'ordre dans lequel les instructions sont exécutées est garanti.
- Il n'est pas possible de faire des choses « en parallèle ».

Exécution




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

Création d'un thread

Première méthode : étendre la classe Thread

- Le code à exécuter doit être implémenté dans la méthode `public void run()` (héritée : surcharge).
- La méthode `start()` héritée de la classe démarre le thread.

```
public class MonThread extends Thread{
    public void run() {
        System.out.println("bonjour !");
    }
}
public class Threads {
    public static void main(String[] args) {
        MonThread T = new MonThread();
        T.start();
    }
}
```

 **créer un nouveau fil d'exécution**

Comment faire pour hériter d'une autre classe ?

Création d'un thread

Deuxième méthode : créer un objet qui implémente l'interface **Runnable**

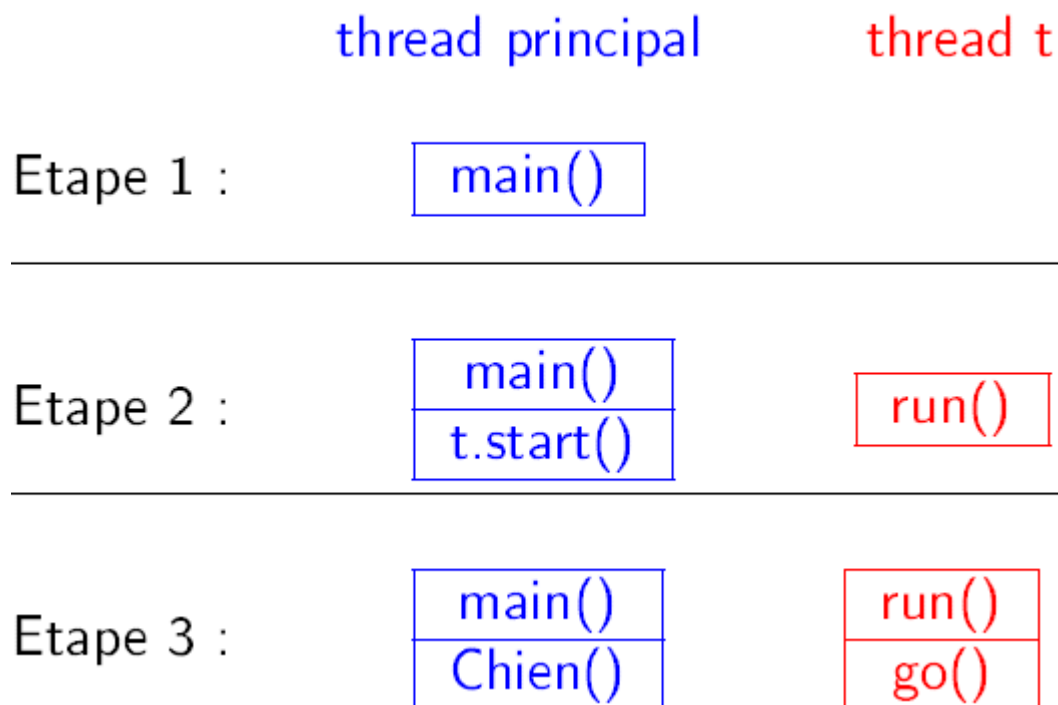
- Le code à exécuter doit être implémenté dans la méthode `public void run()` (obligatoire)
- On passe l'objet en paramètre du constructeur d'un thread.

```
public class MonThreadRunnable implements Runnable{
    public void run(){
        System.out.println("bonjour from Runnable!");
    }
}
```

```
public class Threads {
    public static void main(String[] args) {
        MonThreadRunnable TR = new MonThreadRunnable();
        Thread t = new Thread(TR);
        t.start();
    }
}
```

Exemple

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



Etats d'un thread

- Dans un environnement mono-thread, lorsqu'un thread se bloque (voit son exécution suspendue) en attente d'une ressource, le programme tout entier s'arrête. Ceci n'est plus le cas avec un environnement multi-threads
- Un thread peut être :
 - * en train de s'exécuter (running) ;
 - * prêt à s'exécuter (ready to run), dès que la CPU est disponible ;
 - * suspendu (suspended), ce qui stoppe temporairement son activité ;
 - * poursuivre l'exécution (resumed), là où il a été suspendu ;
 - * bloqué (blocked) en attendant une ressource.

Un thread peut se terminer à tout moment, ce qui arrête son exécution immédiatement. Une fois terminé, un thread ne peut être remis en route.

Le thread et ses 7 etats

1) nouveau (new) : Lorsqu'un nouveau thread est créé, par exemple avec

```
Thread th = new Thread(a);
```

2) prêt (runnable) : Lorsque la méthode `start()` d'un thread est appelée, ce dernier passe dans l'état prêt. Tous les threads prêts d'un programme Java sont organisés par la machine virtuelle en une structure de données nommée la file d'attente prête. Un thread entrant dans l'état prêt est mis dans cette file. Le code de la méthode `run()` sera appelé à l'exécution du thread.

3) en cours d'exécution (running) : Le thread se voit allouer des cycles CPU par l'ordonnanceur. Si un thread dans l'état en cours d'exécution appelle sa méthode `yield()`, ou s'il est supplanté par un thread de priorité plus élevée, il laisse la CPU et est remis dans la file d'attente prête, entrant dans l'état prêt.

4) suspendu (suspended) Un thread prêt ou en cours d'exécution entre dans l'état suspendu lorsque sa méthode `suspend()` est appelée. Un thread peut se suspendre lui-même ou l'être par un autre. De l'état suspendu, un thread ré-entre dans l'état prêt lorsque sa méthode `resume()` est appelée par un autre thread.

5) bloqué (blocked) Un thread entre dans l'état bloqué lorsque :

- * il appelle sa méthode `sleep()`,
- * il appelle `wait()` dans une méthode synchronisée d'un objet,
- * il appelle `join()` sur un autre objet dont le thread ne s'est pas encore terminé,
- * il effectue une opération d'entrée/sortie bloquante (comme lire à partir du clavier).

À partir de l'état bloqué, un thread ré-entre dans l'état prêt.

6) suspendu-bloqué (suspended-blocked) Lorsque un thread bloqué est suspendu par un autre thread. Si l'opération bloquante se termine, le thread entre dans l'état suspendu. Si le thread est réintégré (resumed) avant que l'opération bloquante ne se termine, il entre dans l'état bloqué.

7) mort (dead) Un thread se termine et entre dans l'état mort lorsque sa méthode `run()` se termine ou lorsque sa méthode `stop()` est appelée.

Actions sur d'un thread

- **start()**: démarre l'exécution d'un thread
- **yield()**: redonne explicitement le contrôle au scheduler.
- **sleep(long)**: fait passer le thread en état d'attente pendant une certaine durée (comptée en millisecondes).
- **setPriority(int)** : changer la priorité du thread. Le thread avec la plus grande valeur (le plus prioritaire) est activé préférentiellement.
- **join()** : appliquée sur un objet t de type Thread permet d'attendre la fin de l'exécution de t : `t.join(); // attendre la fin de t`

Actions sur d'un thread

- **run()**: la méthode qui est exécutée lorsque le thread s'exécute. C'est souvent une boucle.
- **stop()**: termine son exécution (il passe dans l'état "mort").
- **suspend()**: suspend l'exécution (doit être relancé par un "resume" ensuite. Fait passer le thread dans un état d'attente (not runnable).
- **resume()**: relance et "résume" l'exécution.

Exercice

Ecrire un programme avec deux threads, qui correspondent chacun à un compteur. L'un des compteurs incrémentera une valeur initiale (init) d'un incrément (inc) positif, et ce pour un nombre d'itérations donné (Nit). Ce thread sera interrompu de « x » temps entre chaque deux itérations. Pareil pour le deuxième compteur, mais avec un incrément (inc) négatif. Le temps d'interruption ne sera pas le même pour les deux threads. Commenter le résultat obtenu.

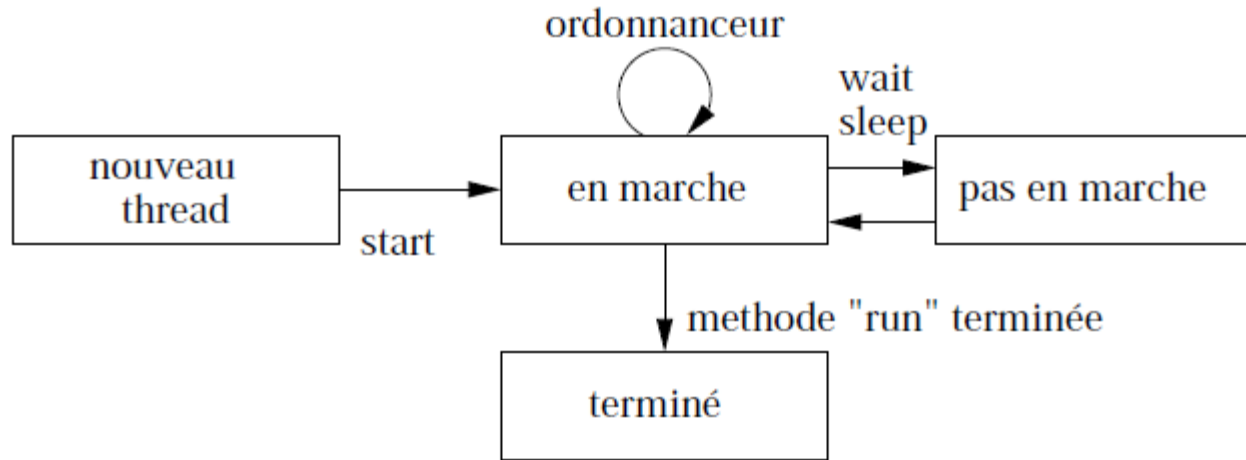
```

class Compteur extends Thread {
protected int count, inc, delay, Nit;
public Compteur(int init, int inc, int delay, int Nit){
    this.count=init;
    this.inc=inc;
    this.delay=delay;
    this.Nit = Nit;
}
public void run(){
    try{for(int i=0; (i < Nit); i++){
        System.out.println(count + " ");
        count += inc;
        sleep(delay);
    }
}catch(InterruptedException e){}
}
}

public class Threads {
    public static void main(String[] args) {
        new Compteur(0,1,30,10).start();
        new Compteur(0,-1,100,10).start();
    }
}

```

Cycle de vie d'un thread



- L'appel à la méthode `start` crée une nouvelle instance d'exécution, on arrive dans l'état "en marche" où le fonctionnement du thread est alors soumis à l'ordonnanceur de la MV Java.
- Une première façon de quitter l'état "en marche" est la terminaison de la méthode `run()` qui achève également l'instance d'exécution.
- Il existe deux méthodes pour suspendre le fonctionnement d'un thread :
 - l'appel à la méthode `sleep` où l'exécution du thread est suspendue pour un temps donné.
 - l'appel à la méthode `wait` qui permet d'attendre certains évènements.

Synchronisation

On se place dans le cadre d'une application bancaire où deux personnes accèdent simultanément au même compte. Chaque personne est modélisée par un thread. Les personnes se partagent un objet, une instance de la classe `Compte` définie dans le répertoire `/TP5/banque/`.

Utilisez les classes que vous trouvez dans ce répertoire, analysez les, et essayez d'expliquer le comportement du programme.

Analysons la situation ...

- Le thread `tata` appelle `retirerArgent` avec l'argument 20. Le thread `tata` vérifie alors que la valeur de `montant` est bien inférieur ou égale à celle de `valeur`.
- Le thread `toto` appelle alors `retirerArgent` avec l'argument 20, avant que `tata` ait effectué le retrait. Le thread `toto` vérifie alors que la valeur de `montant` est bien inférieur ou égale à celle de `valeur`.
- On est alors dans une position où les deux threads peuvent simultanément faire un retrait.
- Il est important de comprendre qu'ici on a systématiquement -20 sur le compte à cause de l'appel à la méthode `sleep(700)`.
- Mais même si on enlevait cet appel, le scénario précédent reste possible même si il n'est pas systématique. On comprend dès lors pourquoi les programmes multi-threadés sont difficiles à déboguer !

Solution : la synchronisation (synchronized)

- Pour résoudre ce problème il faut que le test (`montant > valeur`) et l'instruction `valeur = valeur - montant` soient réalisées par le même thread sans être interrompu.
- Un mot clé Java permet de faire cela : `synchronized`. Il suffit par exemple de rajouter dans le prototype de la méthode :

```
public synchronized boolean retirerArgent(int montant,  
String nom) { ... }
```

Faites les modifications nécessaires alors pour corriger ce problème.

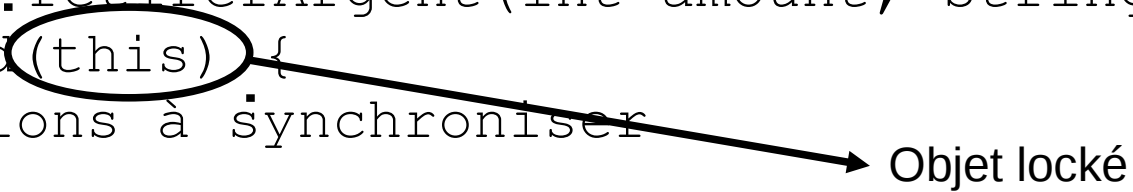
La synchronisation

Remarques :

- Chaque objet Java dispose d'un verrou, utilisé avec le mot-clé `synchronized`: un seul thread à la fois peut appeler la méthode, sinon verrouillage.

- Le mot clé `synchronized` peut également s'utiliser dans le corps de la méthode `retirerArgent` :

```
public boolean retirerArgent(int amount, String nom) {  
    synchronized(this) {  
        // instructions à synchroniser  
    }  
}
```



- Dans un bloc `synchronized{...}` d'un objet `o` :

* `wait()` relache le verrou et se met en attente.

* `notify()` reveille un thread en attente

* `notifyAll()` reveille tous les threads en attente

La synchronisation...

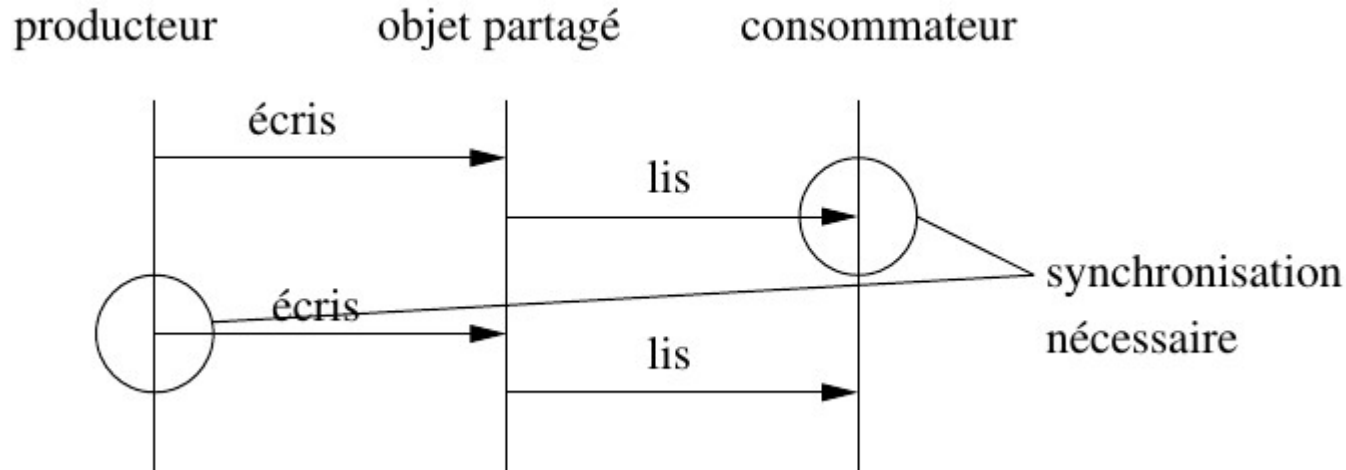
- Chaque objet Java possède un **wait-set** : ensemble des threads qui ont exécuté la méthode `wait()` de cet objet.
- `wait()` : permet au thread qui l'exécute de se mettre en attente dans le wait-set de l'objet récepteur.
- `notify()` : permet à **un thread** de réveiller **un des threads** qui s'était mis en attente dans le wait-set de l'objet récepteur. Il est important de noter qu'un `notify()` n'est pas mémorisé : si le wait-set est vide au moment du `notify()` alors le `notify()` est perdu et aucun thread qui se mettrait plus tard en attente sur le même objet ne pourrait en profiter.
- `notifyAll()` : ressemble à `notify` avec la différence qu'il réveille tous les threads qui était en attente au moment de son exécution.
- Pour assurer la cohérence du wait-set, ces trois méthodes doivent être exécutées dans une section critique verrouillée par l'objet dont on manipule le wait-set.

Communication entre threads

Exemple producteur consommateur

On se place dans le cas du programme multi-threadé suivant : on dispose de deux threads l'un nommé producteur qui écrit dans un objet, l'autre nommé consommateur qui lit dans le même objet qui est donc partagé avec le producteur.

Si le producteur écrit plusieurs fois de suite sans que le consommateur lise, alors les valeurs sont perdues.



Comme montré à la figure ci-dessus il faut donc que le producteur attende que le consommateur ait lu la donnée avant d'en écrire une nouvelle.

L'objet partagé

Une description possible de l'objet partagé sans synchronisation producteur/consommateur :

```
class SharedObject {
    int value=0;
    boolean available = false;
    public SharedObject() { }
    public synchronized void write(int i) {
        if (!available) {
            value=i;
            available=true;
            System.out.println("Ecriture de "+i);
        }
    }
    public synchronized int read() {
        if (available) {
            System.out.println("Lecture de "+value);
            available = false;
            return value;
        }
        return -1;
    }
}
```

L'exécution n'est pas valide car si le résultat n'est pas disponible la méthode `read` renvoie `-1`, il faut alors systématiquement invoquer `read` via `while (read() == -1) { }`.

Une autre solution qui ne marche pas non plus serait dans chaque méthode d'attendre le changement de valeur du paramètre `available`. En effet les méthodes étant `synchronized` elles mettent un verrou sur l'instance de `sharedObject`.

wait et notifyAll

La manière correcte d'implémenter des notions est d'utiliser les méthodes `wait` et `notifyAll`.

L'appel à la méthode `wait` suspend l'exécution d'un thread jusqu'à ce qu'un appel à `notifyAll` soit effectué par un autre thread.

```

class SharedObject {
int value=0;
boolean available = false;
public SharedObject() { }
public synchronized void write(int i) throws InterruptedException {
    if (available) {
        System.out.println("Write en attente");
        wait(); // attendre que le consommateur ait lu
    }
    value=i;
    available=true;
    System.out.println("Ecriture de "+i);
    notifyAll(); // notifier le consommateur
}

public synchronized int read() throws InterruptedException {
    if (!available) {
        System.out.println("Read en attente");
        wait(); // attendre l'écriture du producteur
    }
    System.out.println("Lecture de "+value);
    notifyAll(); //notifier le consommateur que la lecture a été faite
    available = false;
    return value;
}
}

```

Exemple :

Des usagers arrivent à la station pour y attendre un bus ; un bus arrive, charge tous les usagers présents dans la station puis repart. Si un usager arrive trop tard il loupe le bus (pour simplifier le code, on a fait comme si wait() et sleep() ne pouvaient pas déclencher d'exception) :

```
class Station {
    public synchronized void attendreBus      () { wait () ; }
    public synchronized void chargerUsagers  () { notifyAll () ; }
}

class Usager extends Thread {
    private String nom ;
    private Station s ;
    private int heureArrivee ;
    public Usager (String nom, Station s, int heureArrivee) {
        this.nom = nom ;
        this.s = s ;
        this.heureArrivee = heureArrivee ;
    }
    public void run () {
        sleep (heureArrivee) ;
        System.out.println (nom + " arrive a la station") ;
        s.attendreBus () ;
        System.out.println (nom + " est monte dans le bus") ;
    }
}
```

```

class Bus extends Thread {
    private Station s ;
    private int heureArrivee ;
    public Bus (Station s, int heureArrivee) {
        this.s = s ;
        this.heureArrivee = heureArrivee ;
    }
    public void run () {
        sleep (heureArrivee) ;
        System.out.println ("Bus arrive a la station") ;
        s.chargerUsagers () ;
        System.out.println ("Bus repart de la station") ;
    }
}

```

```

class BusSimple {
    public static void main (String args[]) {
        Station Gare = new Station () ;
        Bus b7 = new Bus (Gare, 2000) ;
        Usager u [] = {
            new Usager ("Philippe", Gare, 1500),
            new Usager ("Patricia", Gare, 3000),
            new Usager ("Yves",      Gare, 1000),
            new Usager ("Mireille",  Gare, 1000)
        } ;
        b7.start () ;
        for (int i = 0 ; i < u.length ; i++) u [i].start () ;
    }
}

```


L'exécution donne :

```
Yves arrive a la station  
Mireille arrive a la station  
Philippe arrive a la station  
Bus arrive a la station  
Bus repart de la station  
Yves est monte dans le bus  
Mireille est monte dans le bus  
Philippe est monte dans le bus  
Patricia arrive a la station
```

On est obligé d'arrêter brutalement ce programme : quand Patricia fait son `wait()`, le bus `b7` a déjà fait son `notifyAll()` ; celui-ci n'étant, bien sûr, pas mémorisé, Patricia reste bloquée et empêche l'arrêt du programme !

Priorités

- Les threads peuvent avoir des priorités d'exécutions.
- La priorité d'un thread est l'intervalle: `Thread.MIN_PRIORITY`, `Thread.MAX_PRIORITY`. Les priorités sont valables "en moyenne", c'est à dire qu'en moyenne un thread avec une priorité plus élevée s'exécutera plus souvent.
- `setPriority()` permet de fixer la priorité d'un thread
- Pour 2 threads de même priorité, par défaut : round robin (à chacun son temps)
- T1 cède la place à T2 quand `sleep()`, `wait()`, bloque sur un `synchronized`, `yield()`, `stop()`

Ordonnancement

- L'ordonnateur s'assure que le thread de plus haute priorité (ou les s'il y en a plusieurs d'égale priorité) est exécuté par la CPU.
- Si un thread de plus haute priorité entre dans la file des threads prêts, l'ordonnateur de threads Java met le thread en cours dans la file d'attente, pour que la prioritaire s'exécute.
- Le thread en cours est dit supplantée par préemption par le thread de plus haute priorité.
- Si le thread en cours d'exécution passe la main (via `yield()`, est suspendu ou bloqué, l'ordonnateur choisit dans la file d'attente prête le thread de priorité la plus élevée (ou l'un d'entre eux si ils sont plusieurs) pour être exécutée.

Groupes de threads

-Regroupement des threads par un ThreadGroup.

-Appliquer un ensemble de threads la même opération : manipuler de manière simultanée tous les éléments du groupe et structurer la sécurité.

-Un thread ne peut changer de groupe.

-Un thread appartient au groupe du thread qui le crée.

-On obtient le groupe d'un thread par l'appel à la méthode `getThreadGroup()`.

-Il existe des constructeurs de Thread pour directement créer un thread dans un groupe :

```
public Thread(ThreadGroup group, Runnable runnable)
```

-Les groupes de threads sont organisées en arbre. La racine de cet arbre est le groupe System (System thread group) ; son enfant est le groupe Main, qui contient le thread dit Main, exécutant la méthode `main()`.

-Le groupe par défaut est le main.

Exercice

Dans cet exercice on va s'intéresser au cas d'un producteur qui produit des objets, qui sont consommées par deux consommateurs différents. Ecrire les classes TabObject, Producteur et Consommateur, ainsi que la classe principale.

Suite en TP ...