Notion de thread (1/2)

La machine virtuelle java (JVM) permet d'exécuter plusieurs traitements en parallèle (en pratique, ils s'exécutent par “tranche” et en alternance sur le processeur).

Ces traitements sont gérés par des threads of control (fil de contrôle), ou plus simplement threads.

Les threads Java sont des processus légers : ils partagent un ensemble de codes, données et ressources au sein d'un processus lourd qui les héberge (ce processus est la JVM).

Avantages des processus légers par rapport aux processus système :

- rapidité de lancement et d'exécution
- partage des ressources système du processus englobant
- simplicité d'utilisation

Notion de thread (2/2)

Intérêts d'une application multi-threads :

- gérer l'exécution de traitements longs sans bloquer les autres

  exemples : - calculs, affichage et gestion des interactions dans une interface graphique
  - lancer des programmes démons qui tournent en tâche de fond

- lancer plusieurs exécutions du même code simultanément sur différentes données

  exemples : - répondre à plusieurs requêtes en même temps dans un serveur
  - traiter deux flux sonores en parallèle

- simuler le parallèlisme nécessaire à certaines classes d'applications

- exploiter la structure multi-processeurs des ordinateurs modernes ?

Inconvénients d'une application multi-threads :

- il faut gérer les problèmes de synchonisation entre threads

- une telle application est difficile à écrire et à débogger

Thread et Runnable (1/3)

En Java, un thread est une instance de la classe **`Thread`** qui implémente l'interface **`Runnable`** dont la méthode **`run()`** décrit le traitement à lancer.

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

Un thread est crée en sous classant la classe **`Thread`** et en redéfinissant sa méthode **`run()`**, ou en instanciant le thread avec un objet **`Runnable`**.

```
new Thread(new Runnable(){
    public void run(){
      ...
    }
});
```

Un **`Thread`** qui n'est plus référencé n'est pas détruit par le garbage collector car il est enregistré par un contrôleur d'exécution.

Le thread principal est celui de la méthode **`main`**.

Thread et Runnable (2/3)

```
public class Perroquet extends Thread{

    private String nom;

    public Perroquet(String nom){
        this.nom = nom;
    }

    public void run(){
        try{
            for(i=0;i<3;i++){
                System.out.println(nom + " est content");
                Thread.sleep(500);
            }
        }
        catch(InterruptedException e){System.out.println(e.getMessage());}
    }

}
```

```
Thread t1 = new Perroquet("jacko");
Thread t2 = new Perroquet("jacki");
```

Thread et Runnable (3/3)

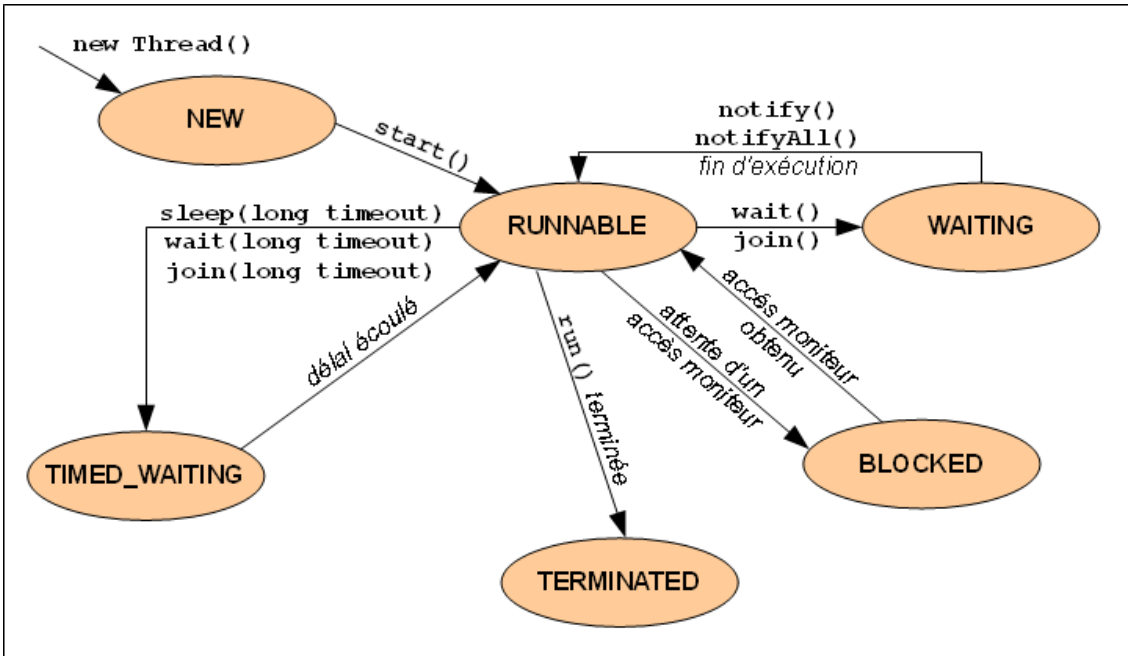
```
public class Perroquet implements Runnable{

    private String nom;

    public Perroquet(String nom){
        this.nom = nom;
    }

    public void run(){
        try{
            for(i=0;i<3;i++){
                System.out.println(nom + " est content");
                Thread.sleep(500);
            }
        }
        catch(InterruptedException e){System.out.println(e.getMessage());}
    }

}
```

```
Thread t1 = new Thread(new Perroquet("jacko"));
Thread t2 = new Thread(new Perroquet("jacki"));
```

on peut créer plusieurs threads différents sur un même objet Runnable

```
Perroquet p = new Perroquet("jacko");
Thread t1 = new Thread(p);
Thread t2 = new Thread(p);
```


Vie et mort d'un thread (1/2)



`destroy`, `resume`, `suspend` et **`stop`** sont des méthodes agissant sur l'état d'un thread, mais qui peuvent poser des problèmes de blocage ou de mauvaise terminaison et sont donc désapprouvées.

Vie et mort d'un thread (2/2)

Activation : la méthode **`start()`** appelle la méthode **`run`**() (**`start()`** n'est pas bloquante dans le thread appelant).

Destruction : elle intervient quand la méthode **`run()`** se termine.

Il est possible d'interrompre le thread en utilisant la méthode **`interrupt()`** qui crée une **`InterruptedException`** si le thread est en attente.

```
public class MonThread extends Thread{

    public void run(){
        try{
            ... // traitement, avec des périodes de sommeil et/ou d'attente
        }
        catch(InterruptedException e){
            ... // libération propre des ressources
        }
        // autre traitement
    }

}
```


Sommeil d'un thread (1/2)

`Thread.sleep(long millis)` est une méthode de classe qui met le thread courant en sommeil un certain nombre de millisecondes. Appelée dans la méthode **`run()`** du thread, elle le met en sommeil.

Si cette méthode est appelée dans du code synchronisé (synchronized) le thread ne perd pas le moniteur (voir plus loin).

L'exécution d'un thread peut également être suspendue par des processus bloquant (entrée/sortie en particulier)

`getState()` renvoie l'état du thread, **`isAlive()`** est vrai si le thread est actif ou endormi

Sommeil d'un thread (2/2)

```
public class TestSleep extends Thread{

    public TestSleep(String name){super(name);}

    public void run(){
        try{
            System.out.println(this.getName()+" a ete lance");
            Thread.sleep(100);
            System.out.println(this.getName()+" est termine");
        }
        catch(InterruptedException e){
            System.out.println(this.getName()+" a ete interrompu");
        }
    }

    public static void main(String arg[]){
        try{
            TestSleep mt = new TestSleep("Toto");
            mt.start();
            Thread.sleep(100);
            mt.interrupt();
        }
        catch(InterruptedException ie){}
    }
}
```

De temps en temps, Toto terminera normalement, d'autres fois il sera interrompu.

La classe Timer

La classe **`java.util.Timer`** permet de lancer un processus une ou plusieurs fois en précisant des délais.

Un **`Timer`** gère les exécutions d'une instance de **`TimerTask`**, classe qui implémente **`Runnable.`**

```
public class ExempleTimer extends TimerTask{

    public void run(){
        try{
            System.out.println("je m'execute");
            Thread.sleep(500);
        }
        catch(InterruptedException e){System.out.println(e.getMessage());}
    }
}
```

```
Timer t = new Timer();
// la tâche se répètera toutes les 2s et démarre dans 1s
t.schedule(new ExempleTimer(),1000,2000);
Date d = new Date();
d.setTime(d.getTime()+10000);
// la tâche démarre dans 10s
t.schedule(new ExempleTimer(),d);
```

Priorités entre threads (1/2)

Une valeur de priorité est affectée à chaque thread et détermine sa priorité d'accès au temps CPU.

Mais la JVM n'assure pas le time slicing : le temps CPU n'est pas forcément partagé équitablement entre threads de même priorité par l'OS.

La priorité est modifiée par **`setPriority(int i)`** et accédée par **`int getPriority()`**:

```
Thread t1 = new Thread(new Perroquet("jacko"));
Thread t2 = new Thread(new Perroquet("jacki"));
t1.setPriority(Thread.MAX_PRIORITY);
t1.start();
t2.start();
```


Priorités entre threads (2/2)

La méthode de classe **`Thread.yield()`** suspend l'exécution du thread qui est en train de s'exécuter pour donner la main à un autre.

- le thread en train de s'exécuter n'est pas mis en sommeil, il est toujours dans la liste des threads actifs
- l'appel de cette méthode peut redonner la main au thread courant!

La méthode **`setDaemon(boolean on)`** appelée avant **`start()`** permet de faire du thread un démon, processus de basse priorité qui tourne en tâche de fond.

exemple : le garbage collector

Synchronisation

Les threads s'exécutant en parallèle sur des données qui peuvent être communes, il faut gérer des conflits d'accès et des problèmes de synchronisation entre threads.

La synchronisation peut consister à entremêler les exécutions des threads de manière à ce qu'ils n'accèdent à certaines données ou à certains morceaux de code que chacun à leur tour alternativement.

Une méthode plus simple est la synchronisation sur terminaison : on veut qu'un morceau de code ne s'exécute qu'après qu'un thread donné ait terminé. La méthode **`join()`** permet une telle synchronisation.

exemple : on veut que jacko attende que jacki ait fini de parler pour prendre la parole

Synchronisation sur terminaison (1/2)

```
public class Perroquet extends Thread{

    private String nom;
    private Perroquet interlocuteur;
    private boolean premier;

    public Perroquet(String nom, boolean b){
        this.nom = nom;
        this.premier = b;
    }

    public void setInterlocuteur(Perroquet p){this.interlocuteur = p;}

    public void run(){
        try{
            if(!premier){
                System.out.println("je vous écoute");
                interlocuteur.join();
                System.out.println("je vous réponds");
            }
            for(int i=0;i<3;i++){
                System.out.println(nom + " est content");
                Thread.sleep(500);
            }
        }
        catch(InterruptedException e){System.out.println(e.getMessage());}
    }

}
```

Synchronisation sur terminaison (2/2)

```
public class Test{


    public static void main(String arg[]){
        Perroquet p1 = new Perroquet("jacko",false);
        Perroquet p2 = new Perroquet("jacki",true);
        p1.setInterlocuteur(p2);
        p2.setInterlocuteur(p1);
        p1.start();
        p2.start();
    }
}
```


La classe Executor

Un **java.util.concurrent.Executor** permet de gérer le lancement d'un ensemble de **Runnable** et la synchronisation sur terminaison.

La classe **java.util.concurrent.Executors** est une « usine » qui fabrique des instances de sous-classes d'**Executor**.

```
// exécution des threads dans une file
ExecutorService ex = Executors.newSingleThreadExecutor();
ex.execute(new Runnable(){
    public void run (){
         ...
    }
});
Runnable r = new MonRunnable();
ex.execute(r);
ex.shutdown();
```

La méthode **Executors.newFixedThreadPool(int nbThreads)** permet de lancer plusieurs threads en même temps mais en nombre limité.

Section critique (1/2)

Un morceau de code est une section critique s'il est nécessaire de garantir qu'au plus un thread exécutera ce code à la fois

```
public class Pile{

    private Object[] tab;private int sommet;

    public Pile(int taille){
        this.tab = new Object[taille];
        this.sommet = -1;
    }

    public void empile(Object o) throws IndexOutOfBoundsException {
        if(this.sommet>=this.tab.length-1) throw new IndexOutOfBoundsException("Pile pleine");
        else{
          this.tab[this.sommet+1] = o;
          this.sommet ++;
        }
    }

    public Object depile() throws IndexOutOfBoundsException {
        if(this.sommet == -1) throw new IndexOutOfBoundsException("Pile vide");
        else{
            this.sommet --;
            return this.tab[this.sommet+1];
        }
    }

}
```

Section critique (2/2)

```
public class Depileur extends Thread{

    private Pile p;

    public Depileur(Pile p){
        super();
        this.p = p;
    }

    public void run(){
        try{
            while(true){
                System.out.println(p.depile());
                Thread.sleep(10);
            }
        }
        catch(IndexOutOfBoundsException e){System.out.println(e.getMessage());}
        catch(InterruptedException e){System.out.println(e.getMessage());}
    }

public static void main(String ar[]){
        Pile p = new Pile(10);
        for(int i = 0;i<10;i++) p.empile(new Integer(i));
        Depileur d1 = new Depileur(p);
        Depileur d2 = new Depileur(p);
        d1.start();
        d2.start();
    }
}
```

Moniteur

Une ressource critique est une ressource qui ne doit être accédée que par un seul thread à la fois.

exemples : variable globale, périphérique de l'ordinateur.

En Java, il n'est pas possible de contrôler directement l'accès simultané à une variable, il faut l'encapsuler et contrôler l'exécution de l'accesseur correspondant.

Un moniteur est un verrou qui ne laisse qu'un seul thread à la fois accéder à la ressource.

En Java, tout objet peut jouer le rôle de moniteur. On pose un moniteur sur un bloc de code à l'aide du mot clé **`synchronized`**

```
synchronized(objetMonitor){
    ... //code en section critique
}
```


Moniteur en Java

- un thread n'accède à la section critique que si le moniteur est disponible
- un thread qui entre en section critique bloque l'accès au moniteur
- un thread qui sort de section critique libère l'accès au moniteur
- **`sleep`** ne fait pas perdre le moniteur (contrairement à **`wait`**)

Attention : le moniteur ne doit pas être modifié dans la section critique! Si le verrou change, la synchronisation n'est plus garantie.

```
synchronized(maListe) {
    maListe = new ArrayList<String>();
}
```

Méthodes synchronisées : on peut déclarer qu'une méthode est en section critique sur le moniteur **`this`**

```
synchronized void methode() {
    //section critique
}
```

```
void methode() {
    synchronized(this) {
        //section critique
    }
}
```

Synchronisation sur moniteur

`wait()` et **`notify()`** synchronisent des threads sur un moniteur :

- l'objet **o** sur lequel les méthodes sont appelées joue le rôle de moniteur
- le thread qui appelle la méthode **`o.wait()`** est placé dans le wait-set de **o**, perd le moniteur et attend
- il redeviendra actif dans un des cas suivants :
  - si la méthode **`o.notify()`** est appelée et qu'il est choisi parmi les threads du wait-set (activation d'un des threads du wait-set, sélection plus ou moins aléatoire)
  - si la méthode **`o.notifyAll()`** est appelée (activation de tous les threads du wait-set)
  - si la durée spécifiée pour le **`wait`** est écoulée (cas où **`wait(int timeout)`** est utilisée)

Le thread qui appelle **`wait`** ou **`notify`** doit posséder le moniteur :

```
synchronized(obj){
    try{
        ...
        obj.wait();
        ...
    }
    catch(InterruptedException e){}
}
```

```
synchronized(obj){
    try{
        ...
        obj.notify();
        ...
    }
    catch(InterruptedException e){}
}
```


Synchronisation des accès à une file (1/2)

```
public class File extends ArrayList<Object>{

    public synchronized void enfiler(Object o){
        System.out.println("enfilage de " + o.toString());
        this.add(o);
        if(this.size()>=1){
            try{
                this.notify();
            }
            catch(InterruptedException e){}
        }
    }

    public synchronized Object defiler(){
        if(this.size()==0){
            try{
                this.wait();
            }
            catch(InterruptedException e){}
        }
        Object o = this.get(0);
        System.out.println("defilage de " + o.toString());
        this.remove(0);
        return o;
   }

}
```

Synchronisation des accès à une file (2/2)

```
public class Enfileur extends Thread{

    private File f;
    private int cpt;

    public Enfileur(File f){
        this.f = f;
        this.cpt = 0;
    }

    public void run(){
        try{
            while(this.cpt < 20){
            f.enfiler("element " + this.cpt);
            Thread.sleep(1000);
            this.cpt ++;
        }
        catch(InterruptedException e){}
   }

}
```

```
public class Defileur extends Thread{

    private File f;

    public Defileur(File f){
        this.f = f;
    }

    public void run(){
        while(true){
            f.defiler();
        }
    }

}
```


Synchronisation et POO

La synchronisation est indépendante de l'héritage :

- une méthode synchronisée peut être redéfinie dans une sous-classe sans être synchronisée.

- une méthode non synchronisée peut être redéfinie et synchronisée dans une sous-classe.

La synchronisation d'une méthode de classe se fait sur l'instance de **`Class`** représentant la classe.

```
public class Machin{

    public static synchronized m(){
        ...
    }

}
```

Sémaphore en Java (1/2)

Un morceau de code est dit réentrant s'il peut être exécuté par plusieurs threads en même temps. Cela suppose que les données sur lesquelles travaille le code soient fournies par les threads. Il peut cependant être nécessaire de limiter le nombre de threads accédant au code.

Un sémaphore permet d'autoriser plusieurs threads à accéder à du code réentrant. Le sémaphore gère un ensemble de permis, en nombre fixé, et accorde ces permis aux threads qui en font la demande.

`java.util.concurrent.Semaphore` implémente ce mécanisme :

- **`Semaphore(int i)`** crée un sémaphore avec i permis initiaux

- la méthode **`acquire()`** permet de requérir un permis et bloque le thread demandeur jusqu'à ce qu'un permis soit disponible ou que le thread soit interrompu. **`acquire(int)`** permet de requérir plusieurs permis.

- la méthode **`release()`** augmente le nombre de permis disponibles de 1. **`release(int i)`** augmente le nombre de permis disponibles de i.

- la méthode **`tryAcquire()`** permet d'acquérir un permis mais n'est pas bloquante.

Sémaphore en Java (2/2)

```
public class MonThread extends Thread{

    private String s;
    private Semaphore sem;

    public MonThread(String s, Semaphore sem){this.s = s;this.sem = sem;}

    public void run(){
        try{
            sem.acquire();
            System.out.println(s+" a un permis ("+sem.availablePermits()+" restants)");
            Thread.sleep(10);
            sem.release();
            System.out.println(s+" n'a plus de permis ("+sem.availablePermits()+" restants)");
        }
        catch(InterruptedException e){System.out.println(e.getMessage());}
    }

    public static void main(String[] a){
        Semaphore sem = new Semaphore(3);
        for(int i = 0;i<5;i++){
            MonThread mt = new MonThread("toto"+i,sem); mt.start();
        }
    }
}
```

La classe Lock (1/2)

Un **`java.util.concurrent.locks.Lock`** permet de réaliser des moniteurs complexes.

Les **`Lock`** permettent une synchonisation plus souple, mais plus exigante que les blocs **`synchronized`**. En particulier, il ne faut pas oublier d'ouvrir le verrou après l'avoir utilisé. Il est conseillé de mettre le code en section critique dans un **`try/finally`**, pour être sur de déverrouiller le **`Lock`** dans tous les cas.

```
Lock l = new ReentrantLock();
l.lock();
try{
    //section critique
}
finally{
    l.unlock();
}
```

Un des avantages de **`Lock`** est la possibilité de tenter d'acquérir le verrou et s'il est déjà pris, de faire autre chose : méthode **`tryLock()`**

La classe Lock (2/2)

Des objets **`Condition`** peuvent être ajoutés au verrou pour faire de la synchronisation sur plusieurs verrous croisés.

```
public class LockExample {

    private Lock lock = new ReentrantLock();
    private Condition cond1  = lock.newCondition();
    private Condition cond2 = lock.newCondition();

    public void m() throws InterruptedException {
    lock.lock();
    try {
        cond1.await();
        ...
        cond2.signal();
    }
    finally {
        lock.unlock();
    }

}
```