

Programmation concurrente en Java

1. Processus et thread

Les unités basiques d'exécution sont généralement les *processus* et les *threads*. Les *processus* possèdent leur propre contexte d'exécution. En particulier, ils possèdent leur propre espace mémoire. Les *threads* sont souvent appelés des *processus légers*. Ils partagent les ressources du processus auquel ils appartiennent. Pour ce TP nous nous intéressons uniquement aux *threads* Java.

2. Objet Thread

En java, chaque thread est associé à une instance de la classe **Thread**. Lorsqu'une application crée un objet **Thread**, elle doit fournir le code à exécuter par le thread. Elle peut le faire de deux manières:

- En fournissant un objet **Runnable**. L'interface **Runnable** définit une unique méthode, **run**, contenant le code à exécuter dans le thread. L'objet **Runnable** est ensuite passé en paramètre du constructeur de l'objet **Thread**:

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

- En étendant directement la classe **Thread** et en fournissant sa propre implémentation de la méthode **run**:

```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

Rappel : méthodes basiques de manipulation des threads

- La méthode **start** place un thread dans l'état **ready**.
- La méthode **sleep** de la classe **Thread** permet de suspendre un thread. Par exemple, **Thread.sleep(1000)** endort le thread courant pendant 1000 ms. L'utilisation de la méthode **sleep** implique de traiter l'exception **InterruptedException**. Cette exception est levée lorsqu'un thread tente d'interrompre l'exécution d'un thread déjà interrompu (par les méthodes **sleep** ou **wait** par exemple). On la traitera de la manière suivante:

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    return;
}
```

Q1. Compiler et exécuter les programmes **HelloRunnable** et **HelloThread**.

Q2. Quel est l'intérêt d'utiliser l'interface **Runnable** plutôt que d'hériter directement de la classe **Thread** ?

3. Synchronisation

Plusieurs threads peuvent accéder aux mêmes objets de manière concurrente. Il est donc nécessaire de mettre en place des mécanismes de synchronisation pour garantir l'accès aux sections *critiques*.

En Java, tout objet possède un verrou. Il s'utilise grâce au mot-clé **synchronized**:

```
synchronized(sharedObj) {  
    // section critique  
}
```

Lorsqu'un thread souhaite entrer dans la section critique, il doit obtenir le verrou de l'objet **sharedObject**. Si un autre thread possède déjà ce verrou, il se met en attente jusqu'à ce que le verrou soit relâché.

Rappel : d'autres mécanismes de synchronisation existent:

- La méthode **join** permet d'attendre la terminaison d'un thread.
- La méthode **wait** s'utilise à l'intérieur des blocs synchronisés. Elle suspend l'exécution du thread courant et le force à rendre le verrou pour les autres threads.
- La méthode **notify** débloque un thread suspendu par **wait**. Aucune garantie n'est fournie sur le thread choisi.
- La méthode **notifyAll** débloque tous les threads suspendus par **wait**.

4. Sémaphore

Un sémaphore correspond à un compteur qui ne peut être accédé que par deux opérations atomiques: acquire et release. L'opération acquire décrémente le compteur alors que l'opération release l'incrémente.

Les sémaphores sont souvent utilisés pour créer des sections critiques accessibles à n threads (n correspondant à la valeur d'initialisation du compteur `permits`). L'opération acquire doit donc mettre en attente le (n+1)ième thread souhaitant entrer en section critique. L'opération release réveille un thread en attente.

Lorsque la variable `permits` est négative, sa valeur absolue représente le nombre de threads en attente.

La classe **Semaphore** vous est fournie pour ce TP :

```
class Semaphore {  
    private int permits;  
    public Semaphore(int initialPermits) {  
        permits = initialPermits;  
    }  
    public synchronized void acquire() {  
        permits = permits - 1;  
        while (permits < 0) {  
            try {  
                wait();  
            } catch (InterruptedException e) {  
            }  
        }  
    }  
    public synchronized void release() {  
        permits = permits + 1;  
        if (permits <= 0)  
            notify();  
    }  
}
```

Q3. Utiliser la classe *Semaphore* pour compléter les classes *Main*, *ThreadA*, *ThreadB* et *ThreadC* fournies. Les classes *ThreadA*, *ThreadB* et *ThreadC* affichent respectivement cinq A, cinq B et cinq C. Synchronisez vos threads à l'aide de sémaphores de sorte que l'affichage effectué par l'application soit : ABCABCABCABCABC.

```
public class Main {
    public static void main(String args[]) {
        Semaphore mutexA = new Semaphore(...);
        ...
        new ThreadA(mutexA, ...).start();
        new ThreadB(...).start();
        new ThreadC(...).start();
    }
}

public class ThreadA extends Thread {
    Semaphore mutexA;
    ...

    public ThreadA(Semaphore mutexA, ...) {
        this.mutexA = mutexA;
        ...
    }

    public void run() {
        for (int i = 0; i < 5; i++) {
            ...
            System.out.print("A");
            ...
        }
    }
}

public class ThreadB extends Thread {
    ...
    public ThreadB(...) {
        ...
    }

    public void run() {
        for (int i = 0; i < 5; i++) {
            ...
        }
    }
}

public class ThreadC extends Thread {
    ...
    public ThreadC(...) {
        ...
    }

    public void run() {
        for (int i = 0; i < 5; i++) {
            ...
        }
    }
}
```

5. CarPark : simulation d'un parking

On souhaite modéliser un parking (classe **CarPark**). Ce parking possède un nombre limité de places défini dans la variable **capacity**. Chaque voiture est modélisée par un thread (classe **Cars**). Chaque voiture essaye d'entrer dans le parking (méthode **arrive**), attend entre 0 et 20 secondes et s'en va (méthode **depart**). La méthode principale est définie dans la classe **CarPark**. Elle crée un parking (c'est à dire une instance de la classe **CarPark**) puis 100 voitures (en espaçant les créations de 0 à 5 secondes).

Q5. Compléter les méthodes **arrive** et **depart** de la classe **CarPark** sans utiliser les sémaphores. La classe **Cars** est fournie.

```
public class Cars implements Runnable {
    private CarPark carpark;
    private Random r;
    private String nom;

    public Cars(String nom, CarPark carpark) {
        this.nom = nom;
        this.carpark = carpark;
        r = new Random();
    }

    public void run() {
        Thread.currentThread().setName(nom);
        carpark.arrive();
        try {
            Thread.sleep(r.nextInt(10) * 1000);
        } catch (InterruptedException e) {
            return;
        }
        carpark.depart();
    }
}

public class CarPark {
    private int capacity;

    public CarPark(int capacity) {
        this.capacity = capacity;
    }

    public synchronized void arrive() {
        System.out.println(Thread.currentThread().getName() + " arrive");
        // completer...
        System.out.println(Thread.currentThread().getName() + " attend...");
        // completer...
        System.out.println(Thread.currentThread().getName()
            + " entre dans le parking [il reste " + capacity + " place(s)");
    }

    public synchronized void depart() {
        System.out.println(Thread.currentThread().getName() + " depart");
        // completer...
    }

    public static void main(String args[]) {
        CarPark carpark = new CarPark(4);
        Random r = new Random();

        for (int i=0; i<100; i++) {
            try {
                Thread.sleep(r.nextInt(5) * 1000);
            } catch (InterruptedException e) {
            }
            new Thread(new Cars("voiture"+i, carpark)).start();
        }
    }
}
```

Question Bonus. Recommencer en utilisant les sémaphores.