Université Grenoble Alpes
U.F.R. Sciences de l'Homme et de la Société
Master MIASHS DCISS

# TP4 : Programmation concurrente en Java

## 1. Processus et thread

Les unités d'exécution de base 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.

Etant donné que la machine virtuelle Java s'exécute généralement au sein d'un unique processus, nous nous intéressons ici uniquement aux *threads* Java.

Tous les programmes que vous manipulerez sont dans un projet Eclipse : dossier tp5 à copier dans votre répertoire workspace à partir du répertoire `/media/commun/tplinux/tp5/workspace` :

```
cp -r /media/commun/tplinux/tp5/workspace/tp5  ~/workspace/tp5
```

Démarrez Elipse et importez le projet (existing project into workspace).

## 2. Objet `Thread` (rappel du cours)

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` définie 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();
    }
}
```

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 (le place dans l'état `sleeping`). 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;
}
```

La méthode `yield` permet de placer un thread en cours d'exécution (dans l'état `running`) dans l'état prêt (`ready`) avant qu'il n'ait terminé son quantum de temps.

**Q1.** Compilez et exécutez 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 (rappels)

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(sharedObject) {
   // 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.

Il est possible de définir des méthodes synchronisées :

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

Dans l'exemple ci-dessus, la classe `SynchronizedCounter` possède trois méthodes synchronisées. Cela garantit que les exécutions de deux de ces méthodes sur la même instance de la classe (le même objet) ne peuvent pas s'entrelacer.

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 (`permits`) qui ne peut être accédé que par deux opérations de base: `acquire` et `release`. L'opération `acquire` décrémente le compteur `permits` alors que l'opération release l'incrémente (voir le cours à ce sujet).

Les sémaphores sont souvent utilisés pour créer des sections critiques accessibles simultanément 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.

**Q3.** Compléter la classe Semaphore fournie pour implémenter le comportement décrit ci-dessus

```
class Semaphore {
      private int permits;
      public Semaphore(int initialPermits) {
             permits = initialPermits;
      }
      public ... void acquire() {
             permits = permits - 1;
             if ...
      }
      public ... void release() {
             permits = permits + 1;
             if ...
      }
}
```

**Q4.** Utiliser votre implémentation de la classe `Semaphore` pour compléter les classes `Main`, `ThreadA`, `ThreadB` et `ThreadC` fournies. Les classes `ThreadA`, `ThreadB` et `ThreadC` sont des threads qui ont pour but d'afficher respectivement cinq 'A', cinq 'B' et cinq 'C'. Il s'agit de synchroniser les threads de sorte que l'affichage corresponde à: 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

On souhaite modéliser un parking (classe `CarPark`). Ce parking possède un nombre limité de places défini lors de l'initialisation de 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 `departs`). 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");
		...
				System.out.println(Thread.currentThread().getName()
					+ " attend...");
		...
		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() + " repart");
		...
	}

	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.