

INFO-H-200

Programmation orientée objet

Séance d'exercices 7
Threads

Université libre de Bruxelles
École polytechnique de Bruxelles

Professeur : Hugues Bersini

2015-2016

Thread

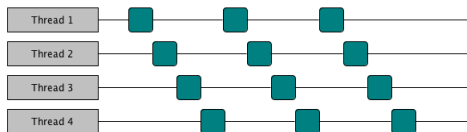
Un Thread est un processus

Le but de l'utilisation des threads est d'exécuter des opérations en parallèle.

Exemple : Un logiciel de traitement de texte vérifie l'orthographe pendant que l'utilisateur tape du texte.

- Un thread qui gère le clavier
- Un thread qui gère la correction orthographique

Pour donner une illusion de parallélisme sur un processeur, l'OS (ou la JVM) exécute les processus et threads de façon concurrente et synchronisée.



Etats : "Nouveau", "Exécutable", "En Attente", "Mort".

Comment faire ?

Deux manières de faire :

- Hériter de la classe Thread (peu pratique)
- Implémenter l'interface Runnable (préféré)

Une classe *thread* doit contenir une méthode *void run()* qui définit le comportement et le cycle de vie de ce thread.

Une fois la méthode *run()* terminée, le thread s'arrête et est détruit.

Pour démarrer le thread, il faut appeler sa méthode *start()*.

Premier Thread via interface Runnable

```
public class MyTimer implements Runnable{
    String string;
    int waitTime;

    public MyTimer(String string, int waitTime){
        this.string = string;
        this.waitTime = waitTime;
    }

    @Override
    public void run(){
        try{
            while(true){
                System.out.print(string);
                Thread.sleep(waitTime);
            }
        } catch(Exception e){};
    }

    public static void main(String[] args){
        Thread t1 = new Thread(new MyTimer("A",100));
        Thread t2 = new Thread(new MyTimer("B",100));
        Thread t3 = new Thread(new MyTimer("C",200));
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Threads sans attente

```
public class MySync implements Runnable{
    private static int counter = 0;

    @Override
    public void run() {
        int temp = counter;
        temp = temp + 1;
        counter = temp;
    }

    public static void main(String args[]) throws InterruptedException{
        Thread t1 = new Thread(new MySync());
        Thread t2 = new Thread(new MySync());
        t1.start();
        t2.start();
        System.out.println("Counter = " + String.valueOf(counter));
    }
}
```

Threads avec attente

```
public class MySync implements Runnable{
    private static int counter = 0;

    @Override
    public void run() {
        int temp = counter;
        temp = temp + 1;
        counter = temp;
    }

    public static void main(String args[]) throws InterruptedException{
        Thread t1 = new Thread(new MySync());
        Thread t2 = new Thread(new MySync());
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Counter = " + String.valueOf(counter));
    }
}
```

Accès concurrents

Comme ce n'est pas le programmeur qui gère l'ordonnancement des threads, il peut arriver que deux threads accèdent à la même ressource en même temps.

Cela peut être dangereux. Pour cela, un thread qui utilise une ressource partagée doit la réserver pour en avoir un accès exclusif.

Pour avoir un accès exclusif à un objet, ou attendre que celui-ci soit disponible, il faut déclarer la méthode comme **synchronized**.

Deadlocks

Situation d'interblocage

Exemple :

- soit deux threads t1 et t2 et deux ressources r1 et r2
- t1 a la main et bloque r1
- t2 prend la main et bloque r2
- t1 prend la main et veut bloquer r2. Il doit attendre que r2 soit libre et donc que t2 la libère
- t2 prend la main et veut bloquer r1. Il doit attendre que r1 soit libre et donc que t1 la libère.
- Le système est bloqué !