JAV – TD 8

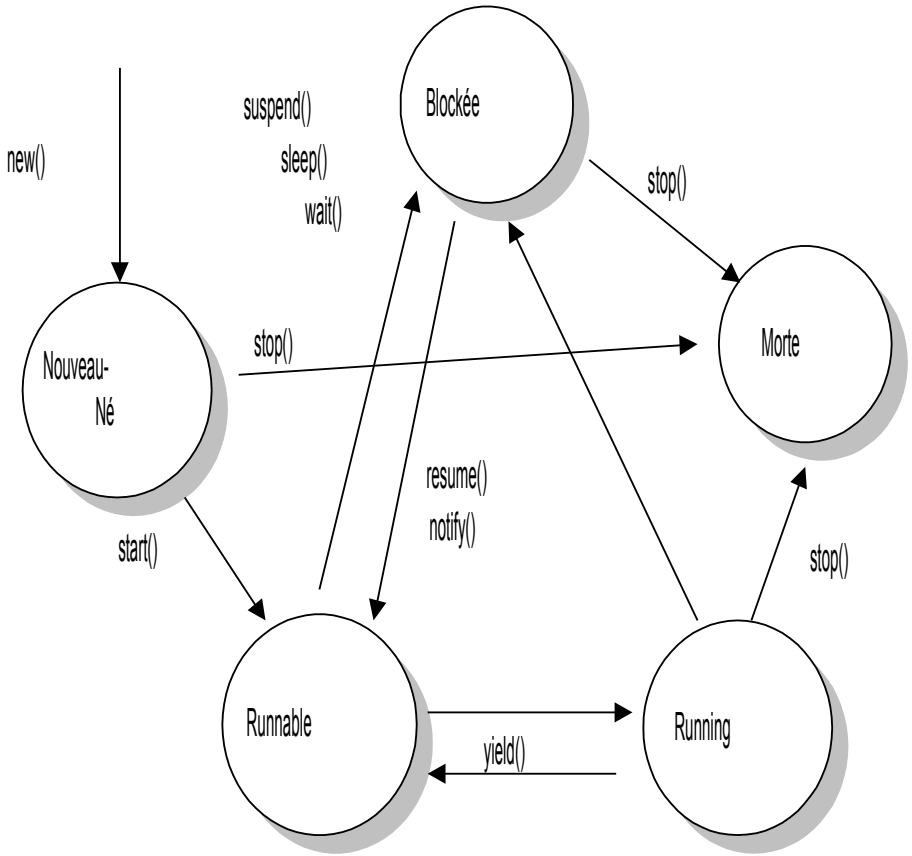
Les threads en Java

Les threads – Objectifs

- Thread = fil d'exécution dans un processus
- Permet au sein d'un même programme (processus) d'exécuter parallèlement plusieurs tâches, de manière indépendante ou de manière synchronisée
- Tout objet Java s'exécute dans au moins un thread

Les threads – Vie et mort

- Exécution de tâches en //
- Mémoire, Code et Ressources partagés
- Économie de ressources
- Un thread ~= méthode qui rend immédiatement la main
- Exemple événements (IHM, GC)
- + priorités
- + synchronisation
  - (moniteur, synchronized)
- Implantation dépendante du SE



Les threads – Syntaxe 1

```
public class Compteur extends Thread {
   public void run() {...}
}

Compteur c = new Compteur();
c.start();
```

- L'héritage à partir de Thread est contraignant car il empêche tout autre héritage

Les threads – Syntaxe 2

```
public class Compteur implements Runnable {
   public void run() {...}
}

Compteur c = new Compteur();
new Thread(c).start();
```

- L'implémentation de l'interface Runnable permet une grande flexibilité : possibilité d'héritage et d'implémentation d'autres interfaces

Exercice

- Écrire le thread Compteur des deux manières possibles
- Tester l'exécution du thread à partir d'un client et à partir de l'objet lui-même

Les threads – Démarrage, arrêt

- `run, start` : démarrage du thread
- `sleep` : pause de x ms du thread
- `yield` : un autre thread peut prendre la main
- `interrup` : interruption du thread

- `isAlive` : le thread est vivant ?
- `isInterrupted` : le thread est interrompu ?

Les threads – Ordonnancement et priorités

- L'ordonnancement est en partie dépendant des implémentations
- Pour 2 threads de même priorité, par défaut : *round robin*
- T1 cède la place a T2 quand `sleep, wait,` bloque sur un `yield`

- `getPriority, setPriority` : récupération ou changement de la priorité d'exécution d'un thread par rapport aux autres

Les threads – Synchronisation

- **Synchronisation physique :**

`synchronized` sur un objet : accès critique à cet objet par un seul thread à la fois

```
public class Compteur implements Runnable {
   private int c;
   public void run() {...}
   public void increment() {
          synchronized(this) { c++; }
   }
}
```

Compteur de nombre pairs

```
public class TestThread implements Runnable{
  int prod=0;
  private int change(){
    this.prod++;
    this.prod++;
    return this.prod;
  }
  public void run(){
    try{while (true){
        System.out.println(">"+this.change());
    }}catch(Exception e){}}
}
```

- 1-Attaquez ce compteur depuis 2 threads (2 threads sur le même objet compteur!).
- 2-Ajoutez un Thread.sleep() entre les 2 incréments. Que se passe-t-il? Pourquoi?

Les threads – Synchronisation

- Synchronisation temporelle :
  - `join` : attend la fin d'un thread
  - `wait` : thread bloqué jusqu'à ce qu'un autre thread appelle `notify` ou `notifyAll` (NB: `wait` libère le verrou)
  - `notify` : débloque un thread bloqué par `wait` (le premier en queue : FIFO)
  - `notifyAll` : débloque tous les threads bloqués par `wait`

Exercice – Producteur/Consommateur

```
/** Example from Sun Thread tutorial */
public class Producer implements Runnable {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        this.cubbyhole = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            this.cubbyhole.put(i);
            System.out.println("Producer #" + this.number
                               + " put: " + i);
            try {
                Thread.sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

Exercice – Producteur/Consommateur

```
/** Example from Sun Thread tutorial */
public class Consumer implements Runnable {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        this.cubbyhole = c;
        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = this.cubbyhole.get();
            System.out.println("Consumer #" + this.number
                                + " got: " + value);
        }
    }
}
```

Exercice – Prod./Cons. – Ressource partagée

```
public class CubbyHole {
    private int contents;

    public int get() {
        return contents;
    }

    public void put(int value)
    {
        contents = value;
    }
}
```

```
Consumer #1 got: 0
Producer #1 put: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Producer #1 put: 1
Producer #1 put: 2
Producer #1 put: 3
Producer #1 put: 4
Producer #1 put: 5
Producer #1 put: 6
Producer #1 put: 7
Producer #1 put: 8
Producer #1 put: 9
```

- Tester l'exécution de cette version du Producteur/Consommateur
- La trace obtenue est-elle correcte ?

Exercice – Prod./Cons. – Ressource partagée

- Modifier la ressource partagée `CubyHole` avec des `synchronized`, `wait()`, `notifyAll()` pour obtenir le comportement correct correspondant à la trace ci-contre

- NB : on ajoutera une variable available qui indique si la valeur est accessible en lecture et/ou écriture

```
Producer #1 put: 0
Consumer #1 got: 0
Producer #1 put: 1
Consumer #1 got: 1
Producer #1 put: 2
Consumer #1 got: 2
Producer #1 put: 3
Consumer #1 got: 3
Producer #1 put: 4
Consumer #1 got: 4
Producer #1 put: 5
Consumer #1 got: 5
Producer #1 put: 6
Consumer #1 got: 6
Producer #1 put: 7
Consumer #1 got: 7
Producer #1 put: 8
Consumer #1 got: 8
Producer #1 put: 9
Consumer #1 got: 9
```

Exercice – Tri fusion

```
trier(debut, fin) {
    si (fin - debut < 2) { // Fin
                        si (t[debut] > t[fin])
                        echanger(t[debut], t[fin])
  }
  sinon {
                        milieu = (debut + fin) / 2
                        trier(debut, milieu)
                        trier(milieu + 1, fin)
                        triFusion(debut, fin) // Tri fusion des moitiés
    }
}
```

Exercice – Tri fusion

- Les deux tris qui sont effectués avant la fusion sont indépendants l'un de l'autre et il est donc “facile” de les faire s’exécuter en parallèle par deux threads
- Implanter une version « threadée » de cet algorithme de tri en les synchronisant avec la méthode `join`
- Implanter une autre version « threadée » de cet algorithme de tri en les synchronisant avec les méthodes `wait` et `notify`