

# Programmation concurrente

# Introduction

- La programmation concurrente correspond à l'ensemble des mécanismes permettant l'exécution concurrente d'actions spécifiées de manière séquentielle.
- En fait, de nombreuses applications fonctionnent simultanément sur vos machines (multitâche).

# Introduction

- Chaque application est associée à un processus.
- Si la machine ne dispose que d'un processeur, l'OS donne l'impression que les applications fonctionnent en même temps en répartissant le temps processeur entre les applications.

# En Java

- On fait la différence entre 2 mécanismes supportant l'ordonnancement automatique des traitements:
  - La concurrence entre commandes du système
  - La concurrence entre processus légers (**thread**) de la JVM

# Thread

- Correspond au fil d'exécution, c'est-à-dire à une suite d'instructions en cours d'exécution.
- Un thread est créé et géré par la JVM.
- Il peut y avoir plusieurs threads pour un programme donné et ils s'exécutent dans un espace mémoire commun (piles différentes mais tas commun).

# Threading en Java

- La méthode `main()` est exécutée par un thread.
- Les applications disposant d'une GUI vont implicitement créées un autre thread, par exemple `AWT event thread`.

# Threading en Java

```
public class MonThread{
    public static void main(String[] args) throws Exception{
        Thread thread1 = Thread.currentThread();
        thread1.setName("Thread du main");
        System.out.println(thread1);
        System.out.println(thread1.isAlive());
        Thread thread2 = new Thread();
        thread2.setName("thread #2");
        System.out.println(thread2);
        System.out.println(thread2.isAlive());
        thread2.start();
        System.out.println(thread2.isAlive());
    }
}
```

1 processus léger appartient à un groupe (ex: main) et a une priorité (ex: 5).

Exécution:

```
Thread[Thread du main,5,main]
true
Thread[thread #2,5,main]
false
true
```

# Création de threads

- Une instance la classe `java.lang.Thread`
- On peut créer une sous classe de `Thread` ou bien créer une classe implémentant l'interface `java.lang.Runnable`.
- On favorise souvent l'approche avec `Runnable`.
- Cette interface dispose d'une seule méthode: `run()`, comme `main()` c'est le point d'entrée du code.
- Si le thread sort de `run()` alors il est considéré comme mort et ne peut redémarrer.

# Démarrage d'un processus léger

- Démarrage d'un processus léger avec la méthode `start()` du thread. La JVM réserve et affecte l'espace mémoire avant d'appeler la méthode `run()` qui exécute le processus léger de la cible.
- Un objet Thread implante l'interface Runnable.
- La méthode `run()` peut donc être invoquée :
  - En implantant la méthode
  - En redéfinissant la méthode de la classe Thread.

# Exemple

**Solution 1: redéfinition de la méthode run() de la classe Thread**

```
class MyThread extends Thread {  
    @Override public void run() { // code à exécuter }  
}  
  
...  
Thread t=new MyThread(); // création  
t.start(); // démarrage  
}
```

**Solution 2 (implantation de l'interface Runnable:**

```
class MyRunnable implements Runnable {  
    public void run() { // code à exécuter }  
}  
  
...  
MyRunnable r=new MyRunnable();  
Thread t=new Thread(r); // création  
t.start(); // démarrage
```

# Création de threads

- Normalement la méthode `run()` n'est pas invoqué explicitement (comme `main()`).
- On passe une instance d'une classe `Runnable` à un thread.
- Exemple:

```
Thread t = new Thread(new myRunnable);
```

- Exemple avec une inner class:

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

# java.lang.Thread

- Pour chaque instance de la classe Thread nous disposons des méthodes:
  - `getName()`
  - `getPriority()` : une valeur entière entre 1 et 10. Plus la valeur est élevée, plus le thread est prioritaire. 3 constantes: `MIN_PRIORITY`, `NORM_PRIORITY`, `MAX_PRIORITY`, resp. 1, 5 et 10.
  - `getThreadGroup()` pour connaître le groupe du thread (classe `ThreadGroup`).

# java.lang.Thread

- `getState()` (depuis Java 1.5) pour connaître l'état d'un processus léger. Expression à l'aide de type énuméré de type `Thread.State`:
  - `NEW` (pas encore démarré)
  - `RUNNABLE` (en cours d'exécution ou attend une ressource)
  - `BLOCKED` (bloqué en attente d'un moniteur(lock))
  - `WAITING` (attente indéfinie d'une ressource d'un autre thread)
  - `TIMED_WAITING` (attente bornée)
  - `TERMINATED`

# java.lang.Thread : sleep

- La méthode (static) `sleep(int)` d'un thread force temporairement l'arrêt de son exécution.
- Exemple, arrêt pendant 1 seconde:  
`Thread.sleep(1000);`

# Sleep exemple

```
public class SleepExample implements Runnable
{
    public void run() {
        long t = System.currentTimeMillis(), x;
        try {
            Thread.sleep(300); // on attend 300ms
        } catch (InterruptedException e) {}
        x = System.currentTimeMillis();
        System.out.println("Attente = "+ (x-t));
    }
    public static void main(String args[]) {
        Runnable r = new SleepExample();
        Thread t = new Thread(r);
        t.start();
    }
}
```

# Caractéristiques des threads

- Lent au démarrage mais une solution de threads pooling peut être utilisé.
- Augmentation de la complexité du code
- Difficulté du partage des ressources entre plusieurs threads.

# Gestion des threads

- Pour permettre à plusieurs opérations une exécution concurrente sur un seul processeur, il est nécessaire de transférer le contrôle du processeur d'un thread à un autre. On parle de *context switching*.
- 2 approches:
  - Coopératif: un thread délaisse volontairement le processeur
  - Préemptif: le contrôle du processeur est passé d'un thread à un autre de manière arbitraire.

# Coopératif vs Préemptif

- Aspect préemptif évite la monopolisation du processeur par un thread, enlève la charge de la gestion du contrôle au programmeur mais ce dernier doit coordonner l'utilisation des ressources partagées par plusieurs threads.
- Avec l'approche préemptive, l'ordre dans d'exécution des threads n'est pas certain. On parle de *race condition*.

# Exemple

```
public class ThreadShare implements Runnable {
    public int partage=0;
    public static void main(String args[]) {
        ThreadShare ts = new ThreadShare();
        Thread t1 = new Thread (ts);
        Thread t2 = new Thread(ts);
        t1.start();
        t2.start();
    }
    public void run() {
        partage+=5;
        System.out.println("Valeur =" + partage);
    }
}
```

Retourne en générale:

Valeur =5

Valeur=10

# Exemple 2

- Mais on pourrait aussi avoir:  
valeur=10; Valeur=10
- Exemple de séquence:
  - t1 entre dans run()
  - t1 ajoute 5 à partage (=5)
  - Context switch
  - t2 entre dans run()
  - t2 ajoute 5 à partage (=10)
  - t2 affiche partage
  - Context switch
  - t1 affiche partage

# Synchronization

- Chaque instance de la classe Object maintient un moniteur (ou lock) et le mot clé `synchronized` est toujours associé avec ces instances.
- Avant qu'un thread utilise une méthode ou portion de codes synchronisée, il doit obtenir le moniteur de cet objet.
- Si le moniteur a été donné à un autre thread alors il doit attendre une libération du thread (situation de blocage).

# Synchronization 2

- En plus du moniteur, chaque objet dispose d'une liste de threads bloqués en attente du moniteur.
- Lorsque le moniteur devient disponible, un des threads de la liste recevra le moniteur et ainsi poursuivra son exécution.

# Exemple 3

```
public class ThreadShare implements Runnable {
    public int partage=0;
    public static void main(String args[]) {
        ThreadShare ts = new ThreadShare();
        Thread t1 = new Thread (ts);
        Thread t2 = new Thread(ts);
        t1.start();
        t2.start();
    }
    public void run() {
        opePartage();
    }
    public synchronized void opePartage() {
        partage +=5;
        System.out.println("Valeur =" + partage);
    }
}
```

# Exemple 4

```
public class ThreadShare2 implements Runnable {
    public Integer partage=0;
    public static void main(String args[]) {
        ThreadShare2 ts = new ThreadShare2();
        Thread t1 = new Thread (ts);
        Thread t2 = new Thread(ts);
        t1.start();
        t2.start();
    }
    public void run() {
        synchronized(partage) {
            partage+=5;
            System.out.println("Valeur =" + partage);
        }
    }
}
```

# Synchronisation 3

- Il faut synchroniser un minimum de ligne de codes.
- Synchronisation de méthode ou bien de bloc de codes.
- Deadlock : un thread t1 attend que le thread t2 libère le moniteur d'un objet mais t2 attend le moniteur bloqué par t1.
  - Solutions: synchronisation de haut niveau ou ordonnancement des blocages.

# Synchronisation entre processus légers

- 2 solutions proposées dans l'environnement Java:
  - `wait()/notify()`
  - `join()`

# Synchronisation entre processus légers

- `wait()` et `notify()` des méthodes la classe `Object`
  - `wait()` pour attendre l'arrivée d'un événement particulier. Le thread doit avoir le moniteur.
  - `notify()` pour indiquer l'arrivée de cet événement.

# Producteur/Consommateur

```
class Q {
int n;
boolean valueSet = false;
synchronized int get() {
    if(!valueSet)
        try {
            wait();
        } catch(InterruptedException e) {
            System.out.println("InterruptedException
caught");
        }
    System.out.println("Got: " + n);
    valueSet = false;
    notify();
    return n;
}
synchronized void put(int n) {
    if(valueSet)
        try {
            wait();
        } catch(InterruptedException e) {
            System.out.println("InterruptedException caught");
        }
    this.n = n;
    valueSet = true;
    System.out.println("Put: " + n);
    notify();
}
}
```

```
class Producer implements Runnable {
    Q q;
    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }
    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}

class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) {
            q.get();
        }
    }
}

public class ProdCons {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);
    }
}
```

# Synchronisation entre processus légers

- `join()` est une méthode de la classe `Thread`
- Elle permet d'attendre d'une terminaison
- `join()` appelé sur l'objet contrôlant un processus léger dont la terminaison est attendue. Le processus courant est interrompu jusqu'à la fin du processus.

# Join, exemple

```
class Slave implements Runnable {
    private int value = 0;
    public int getValue() {
        return value;
    }
    public void run() {
        for(;;value <5;value++) {
            try {
                Thread.sleep(200);
            }
            catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

```
class Master implements Runnable {
    public void run() {
        Slave slave = new Slave();
        Thread slaveThread = new Thread(slave);
        slaveThread.start();
        try {
            slaveThread.join();
        }
        catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
        System.out.println(slave.getValue());
    }
}
```

```
public class SlaveMaster {
    public static void main(String[] args)
    {
        Master master = new Master();
        Thread masterThread = new
        Thread(master);
        masterThread.start();
    }
}
```

Affiche 0 sans le bloc  
Affiche 5 avec le bloc.

