Université Moulay Ismaïl
École Normale Supérieure de Meknès
Département des sciences

Programmation oriontée objet en JAVA

CH5 (Threads)

Pr. Mehdi Moukhafi

Les threads

Qu'est-ce qu'un Thread ?

les threads sont différents des processus :

- ils partagent code, données et ressources : « processus légers »
- mais peuvent disposer de leurs propres données.
- ils peuvent s'exécuter en "parallèle"

Avantages :

- légèreté grâce au partage des données
- meilleures performances au lancement et en exécution
- partage les ressources système (pratique pour les I/O)

Utilité :

- puissance de la modélisation : un monde multithread
- puissance d'exécution : paralèllisme
- simplicité d'utilisation : c'est un objet Java (java.lang)

Création

La classe java.lang.Thread permet de créer de nouveaux threads

Un thread doit implémenter obligatoirement l'interface **Runnable**

- le code exécuté se situe dans sa méthode run()

2 méthodes pour créer un Thread :

- 1) une classe qui **dérive** de java.lang.Thread
  - java.lang.Thread implémente Runnable
  - il faut redéfinir la méthode run()
- 2) une classe qui **implémente** l'interface Runnable
  - il faut implémenter la méthode run()

Méthode 1 : Sous-classer Thread

```
class Proc1 extends Thread {
Proc1() {...} // Le constructeur
...
public void run() {
... // Ici ce que fait le processus : boucle infinie
}
}
...
Proc1 p1 = new Proc1(); // Création du processus p1
p1.start(); // Demarre le processus et execute p1.run()
```

Méthode 2 :
une classe qui implémente Runnable

```
class Proc2 implements Runnable {
Proc2() { ...} // Constructeur
...
public void run() {
... // Ici ce que fait le processus
}
}
...
Proc2 p = new Proc2();
Thread p2 = new Thread(p);
...
p2.start(); //Démarre un processus qui execute p.run()
```

Quelle solution choisir ?

Méthode 1 : sous-classer Thread

- lorsqu'on désire paralléliser une classe qui n'hérite pas déjà d'une autre classe (attention : héritage simple)
- cas des applications autonomes

Méthode 2 : implémenter Runnable

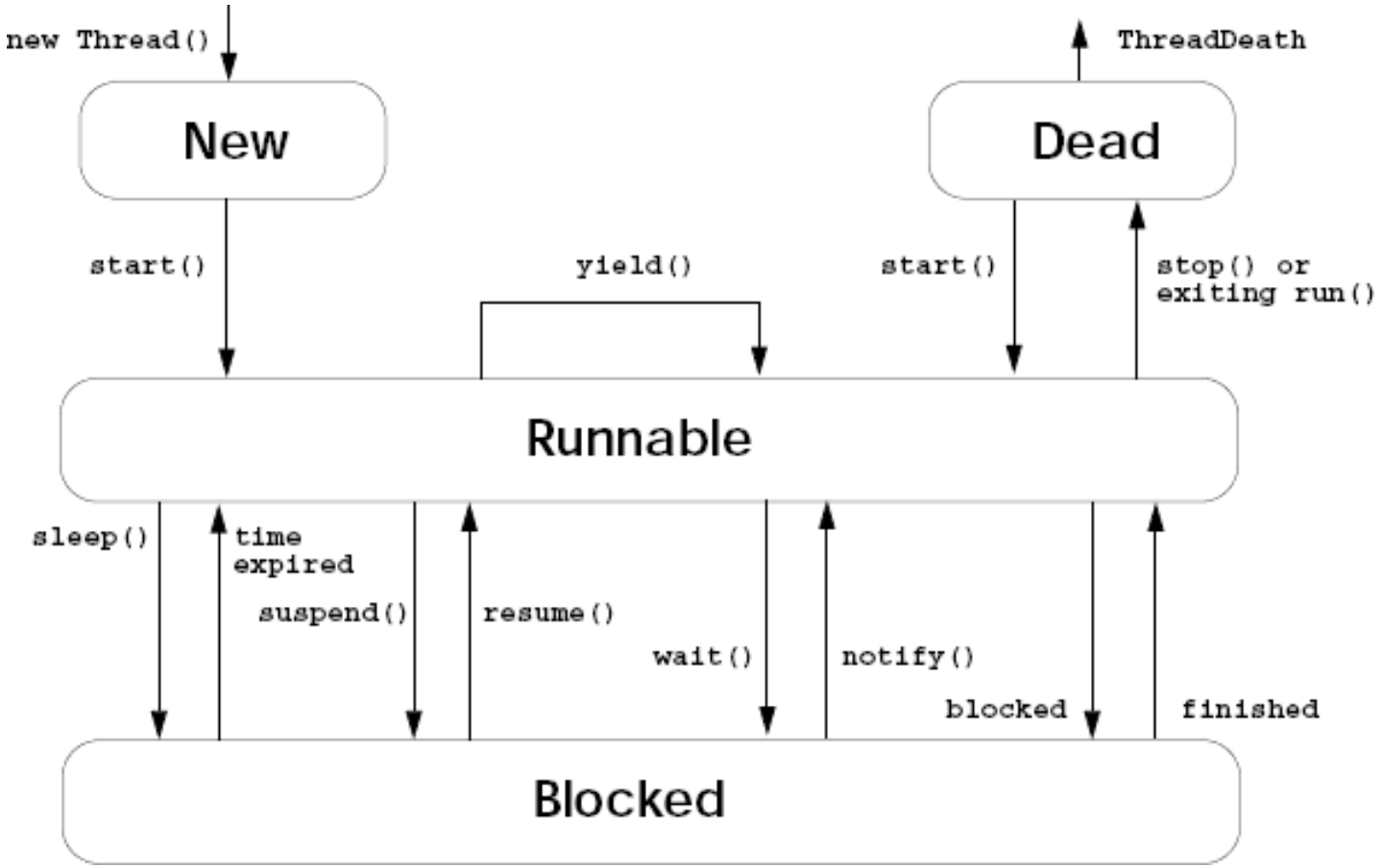
- lorsqu'une super-classe est imposée
- cas des applets

```
public class MyThreadApplet
extends Applet implements Runnable {}
```

Distinguer la méthode **run** *(qui est le code exécuté par l'activité) et la méthode* **start** *(méthode de la classe Thread qui rend l'activité exécutable) ;*

Dans la première méthode de création, attention à définir la méthode run avec strictement le prototype indiqué (il faut redéfinir Thread.run et non pas la surcharger).

Le cycle


new Thread()
New
ThreadDeath
Dead
start()
yield()
start()
stop() or
exiting run()
Runnable
sleep()
time
expired
suspend()
resume()
wait()
notify()
blocked
finished
Blocked

Les états d'un thread

Créé :

- comme n'importe quel objet Java
- ...mais n'est pas encore actif

- **Actif :**
  - après la création, il est activé par start() qui lance run().
  - il est alors ajouté dans la liste des threads actifs pour être exécuté par l'OS en temps partagé
  - peut revenir dans cet état après un resume() ou un notify()

Exemple

```
class ThreadCompteur extends Thread {
int no_fin;
ThreadCompteur (int fin) {no_fin = fin;} // Constructeur
// On redéfinit la méthode run()
public void run () {
for (int i=1; i<=no_fin ; i++) {
  System.out.println(this.getName()+":"+i);} }

public static void main (String args[]) {
// On instancie les threads
ThreadCompteur cp1 = new ThreadCompteur (100);
ThreadCompteur cp2 = new ThreadCompteur (50);
cp1.start();
cp2.start();
} }
```

Les états d'un Thread (suite)

Endormi ou bloqué :

- après **sleep()** : endormi pendant un intervalle de temps (ms)
- **suspend()** endort le Thread mais **resume()** le réactive
- une entrée/sortie bloquante (ouverture de fichier, entrée clavier) endort et réveille un Thread

Mort :

- si **stop()** est appelé explicitement
- quand **run()** a terminé son exécution

Exemple d'utilisation de sleep

```
class ThreadCompteur extends Thread {
int no_fin; int attente;
ThreadCompteur (int fin,int att) {
  no_fin = fin; attente=att;}
public void run () {       //redéfinir run
for (int i=1; i<=no_fin ; i++) {
  System.out.println(this.getName()+":"+i);
  try {sleep(attente);}
      catch(InterruptedException e) {};}
}
public static void main (String args[]) {
// On instancie les threads
ThreadCompteur cp1 = new ThreadCompteur (100,100);
ThreadCompteur cp2 = new ThreadCompteur (50,200);
cp1.start();
cp2.start();
} }
```

Les priorités

Principes :

- Java permet de modifier les priorités (niveaux absolus) des Threads par la méthode setPriority()
- Par défaut, chaque nouveau Thread a la même priorité que le Thread qui l'a crée
- Rappel : seuls les Threads actifs peuvent être exécutés et donc accéder au CPU
- La JVM choisit d'exécuter le Thread actif qui a la plus haute priorité : priority-based scheduling
- si plusieurs Threads ont la même priorité, la JVM répartit équitablement le temps CPU (time slicing) entre tous : round-robin scheduling

Les priorités (suite)

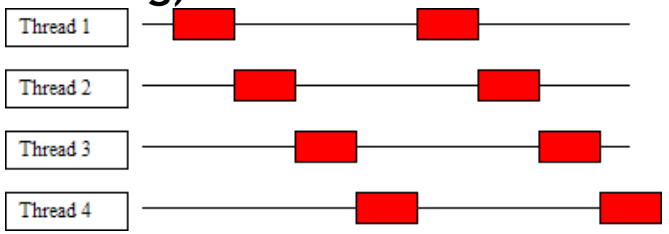
Les méthodes :

- **setPriority(int)** : fixe la priorité du receveur.
  - le paramètre doit appartenir à : [MIN_PRIORITY, MAX_PRIORITY]
  - sinon IllegalArgumentException est levée
- **int getPriority()** : pour connaître la priorité d'un Thread
- NORM_PRIORITY : donne le niveau de priorité "normal"

La gestion du CPU

Time-slicing (ou round-robin scheduling) :

- La JVM répartit de manière équitable le CPU entre tous les threads de même priorité. Ils s'exécutent en "parallèle".



Préemption (ou priority-based scheduling) :

- Le premier thread du groupe des threads à priorité égale monopolise le CPU. Il peut le céder :
  - involontairement : sur entrée/sortie
  - volontairement : appel à la méthode statique yield()

  Attention : ne permet pas à un thread de priorité inférieure de s'exécuter (seulement de priorité égale)
  - implicitement en passant à l'état endormi (wait(), sleep() ou suspend())

Daemons

Un thread peut être déclarer comme daemon :

- comme le "garbage collector", l'"afficheur d'images", ...
- en général de faible priorité, il "tourne" dans une boucle infinie
- arrêt implicite dès que le programme se termine

Les méthodes :

- setDaemon() : déclare un thread daemon
- isDaemon() : ce thread est-il un daemon ?

Les « ThreadGroup »

Pour contrôler plusieurs threads

Plusieurs processus (Thread) peuvent s'éxécuter en même temps, il serait utile de pouvoir les manipuler comme une seule entité

- pour les suspendre
- pour les arrêter,...

Java offre cette possibilité via l'utilisation des groupes de threads :

`java.lang.ThreadGroup`

on groupe un ensemble nommé de threads

ils sont contrôlés comme une seule unité

Les groupes de threads

Une arborescence :

- la classe **`ThreadGroup`** permet de constituer une arborescence de Threads et de ThreadGroups
- elle donne des méthodes classiques de manipulation récursives d'un ensemble de threads : `suspend()`, `stop()`, `resume()`, …
- et des méthodes spécifiques : `setMaxPriority()`, …

Fonctionnement :

- la JVM crée au minimum un groupe de threads nommé main
- par défaut, un thread appartient au même groupe que celui qui l'a crée (son père)
- **`getThreadGroup()`** : pour connaitre son groupe

Création d'un groupe de threads

Pour créer un groupe de processus :

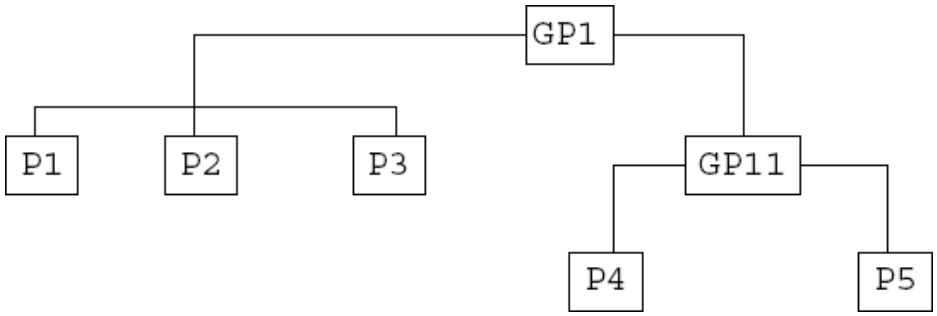
```
ThreadGroup groupe = new ThreadGroup("Mon groupe");

Thread p1 = new Thread(groupe, "P1");
Thread p2 = new Thread(groupe, "P2");
Thread p3 = new Thread(groupe, "P3");
```

On peut créer des sous-groupes de threads pour la création d'arbres sophistiqués de processus

- des ThreadGroup contiennent des ThreadGroup
- des threads peuvent être au même niveau que des ThreadGroup

Création de groupe de threads (suite)

```
ThreadGroup groupe1 = new ThreadGroup("GP1");
Thread   p1 =  new  Thread(groupe1,     "P1");
Thread   p2 =  new  Thread(groupe1,     "P2");
Thread   p3 =  new  Thread(groupe1,     "P3");
ThreadGroup    groupe11 = new ThreadGroup(groupe1,          "GP11");
Thread p4 =    new Thread(groupe11, "P4");
Thread p5 =    new Thread(groupe11, "P5");
```



Contrôler les ThreadGroup

Le contrôle des ThreadGroup passe par l'utilisation des méthodes standards qui sont partagées avec Thread :

`resume(), suspend(), stop(), ...`

- Par exemple : appliquer la méthode `stop()` à un ThreadGroup revient à invoquer pour chaque Thread du groupe cette même méthode
- ce sont des méthodes de manipulation récursive

Avantages /Inconvénients des threads

Programmer facilement des applications où des traitements se résolvent de façon concurrente (applications réseaux, par exemple)

Améliorer les performances en optimisant l'utilisation des ressources

Code plus difficile à comprendre, peu réutilisable et difficile à débuguer