Le langage JAVA

Les threads ou l'exécution parallèle

MUTI-TÂCHES ET PARALLÉLISME

- **Multi-tâches : exécution de plusieurs processus simultanément.**
  - ✓ Un processus est un programme en cours d'exécution.
  - ✓ Le système d'exploitation distribue le temps CPU entre les processus
- **Un processus peut être dans différents états.**
  - ✓ En exécution (running) : il utilise le processeur
  - ✓ Prêt (ready) : le processus est prêt à s'exécuter, mais n'a pas le processeur (occupé par un autre processus en exécution)
  - ✓ Bloqué
- **Parallélisme : pouvoir faire exécuter plusieurs tâches à un ordinateur avec plusieurs processeurs.**
- **Si l'ordinateur possède moins de processeurs que de processus à exécuter :**
  - ✓ division du temps d'utilisation du processeur en tranches de temps (« time slice » en anglais)
  - ✓ attribution des tranches de temps à chacune des tâches de façon telle qu'on ait l'impression que les tâches se déroulent en parallèle.
  - ✓ on parle de pseudo-parallélisme
- **Les systèmes d'exploitation modernes gèrent le muti-tâches et le parallélisme**

ET QU'EST-CE QU'UN THREAD ?

- **Les threads sont différents des processus :**
  - ✓ Ils simulent le fonctionnement de plusieurs processus en parallèle
  - ✓ MAIS dans un seul
  - ✓ Ils partagent code, données et ressources : « processus légers »
  - ✓ ET peuvent disposer de leurs propres données.
  - ✓ Ils sont en fait comme 2 pointeurs dans le programme
- **Avantages :**
  - ✓ légèreté grâce au partage des données
  - ✓ meilleures performances au lancement et en exécution
  - ✓ partage des ressources système (pratique pour les I/O)
- **Utilité :**
  - ✓ puissance de la modélisation : un monde multithread
  - ✓ puissance d'exécution : paraléllisme
  - ✓ simplicité d'utilisation : c'est un objet Java (**java.lang**)

LA 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 {
  // Le constructeur
  Proc1() {...}
  ...
  public void run() {
  ... // Ici ce que fait le processus
  ... //boucle infinie
  }
}
...
// Création du processus p1
Proc1 p1 = new Proc1();
// Démarre le processus et exécute p1.run()
p1.start();
```

MÉTHODE 2 : UNE CLASSE QUI IMPLÉMENTE RUNNABLE

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

QUELLE SOLUTION CHOISIR ?

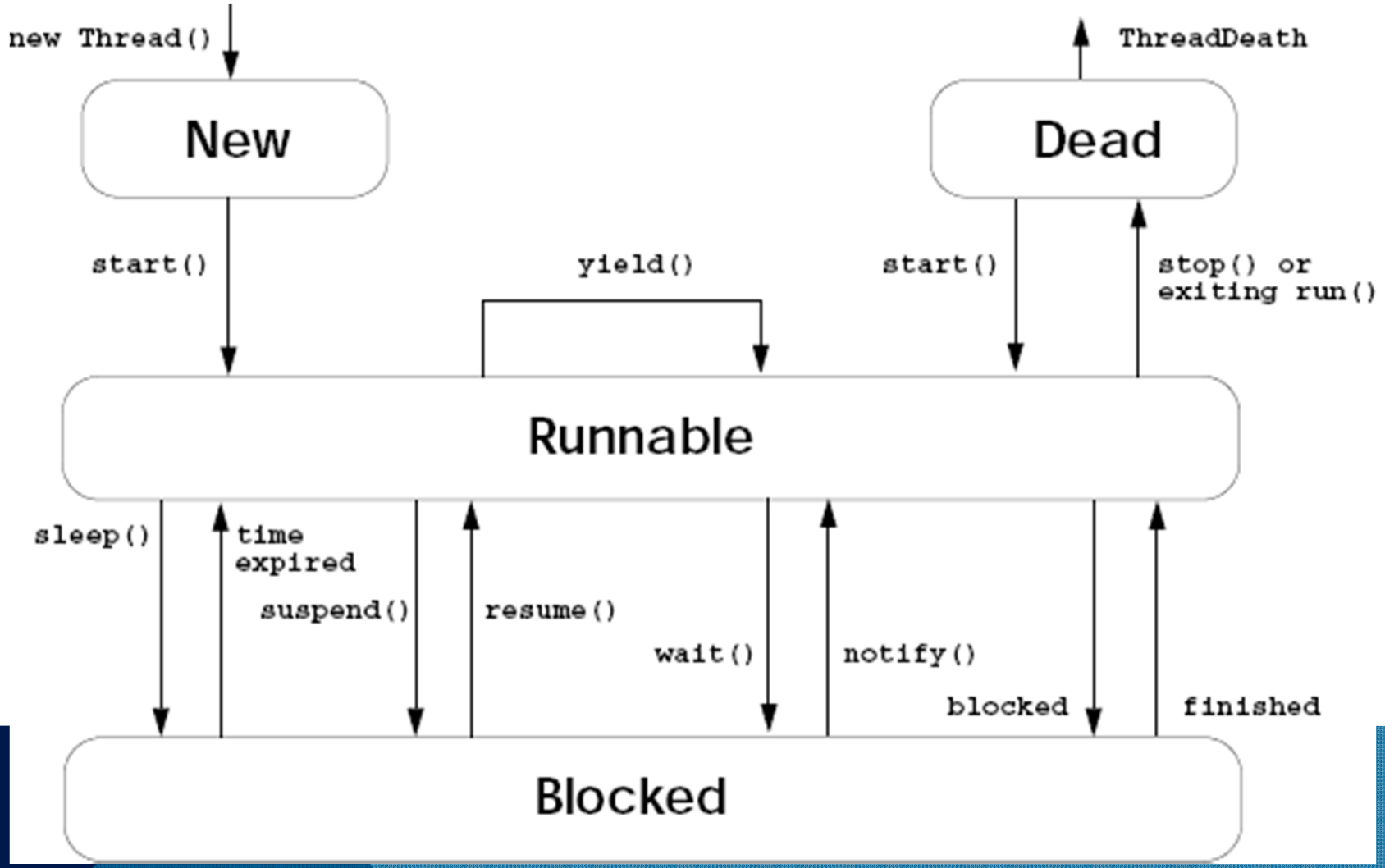
- **Méthode 1 : sous-classer Thread**
  - ✓ lorsqu'on désire paralléliser une classe qui n'hérite pas déjà d'une autre classe (attention : héritage simple)
  - ✓ cas des applications autonomes

- **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 DE VIE



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
  - ✓ Il peut revenir dans cet état après un resume() ou un notify()
- **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 DE CRÉÉ / ACTIF

```
class ThreadCompteur extends Thread {
  int no_fin;

  ThreadCompteur (int fin) { // Constructeur
    no_fin = fin;
  }

  // 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();
    }
}
```

EXEMPLE D'UTILISATION DE SLEEP

```
class ThreadCompteur extends Thread {
  int no_fin;
  int attente;

  ThreadCompteur (int fin,int att) {
    no_fin = fin;
    attente=att;
  }
  // On redéfinit la méthode run()
  public void 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();
  }
}
```

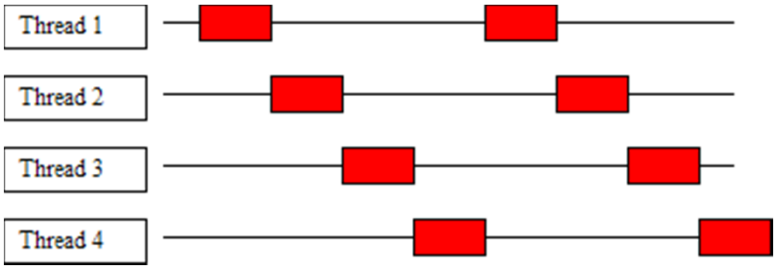
LES PRIORITÉS

- **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 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())

LA CONCURRENCE D'ACCÈS

- **Le problème : l'espace de travail étant commun, il n'y a pas de "mémoire privée" pour chaque thread :**
  - ✓ *Inconvénient* : accès simultané à une même ressource
    Il faut garantir l'accès exclusif à un objet pendant l'exécution d'une ou plusieurs instructions

- **Pour se faire : le mot-clé synchronized permet de gérer les concurrence d'accès :**
  - ✓ d'une méthode
  - ✓ d'un objet
  - ✓ ou d'une instruction (ou d'un bloc)

LA SYNCHRONISATION

- **Basée sur la technique de l'exclusion mutuelle :**
  - ✓ à chaque objet Java est associé un « verrou » géré par le thread quand une méthode (ou un objet) **synchronized** est accédée.
  - ✓ Elle garantit l'accès exclusif à une ressource (la section critique) pendant l'exécution d'une portion de code.
- **Une section critique :**
  - ✓ une méthode : déclaration précédée de **synchronized**
  - ✓ une instruction (ou un bloc) : précédée de **synchronized**
  - ✓ un objet : le déclarer **synchronized**
- **Attention à l'inter-blocage !!**
  - ✓ Lorsque tous les processus attendent les autres
  - ✓ exemple du problème du dîner des philosophes)

UTILISER SYNCHRONIZED

- **Pour gérer la concurrence d'accès à une méthode :**
  - ✓ si un thread exécute cette méthode sur un objet, un autre thread ne peut pas l'exécuter pour le même objet
  - ✓ en revanche, il peut exécuter cette méthode pour un autre objet

```
public synchronized void maMethode() {...}
```

- **Pour Contrôler l'accès à un objet :**

```
public void maMethode() { ...
        synchronized(objet) {
            objet.methode();}}
```

  - ✓ l'accès à l'objet passé en paramètre de **synchronized(Object)** est réservé à un unique thread.

EXEMPLE DE SYNCHRONISATION

```
class Impression {
  // Essayer avec et sans synchronized
  synchronized public void imprime(String t) {
    for (int i=0; i<t.length(); i++) {
      System.out.print(t.charAt(i));
    }
  }
}

class TPrint extends Thread {
  static Impression mImp = new Impression();
  String txt;
  public TPrint(String t) {txt = t;}

  public void run() {
    for (int j=0; j<50; j++) { mImp.imprime(txt);}
  }

  static public void main(String args[]) {
    TPrint a = new TPrint("bonjour ");
    TPrint b = new TPrint("au revoir ");
    a.start();
    b.start();
  }
}
```

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 GROUPES DE THREAD / « THREADGROUP »

- Les groupes permettent de contrôler plusieurs threads
- Plusieurs processus (Thread) peuvent s'exécuter en même temps, il est alors 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 et ils sont contrôlés comme une seule unité

LES GROUPES DE THREADS

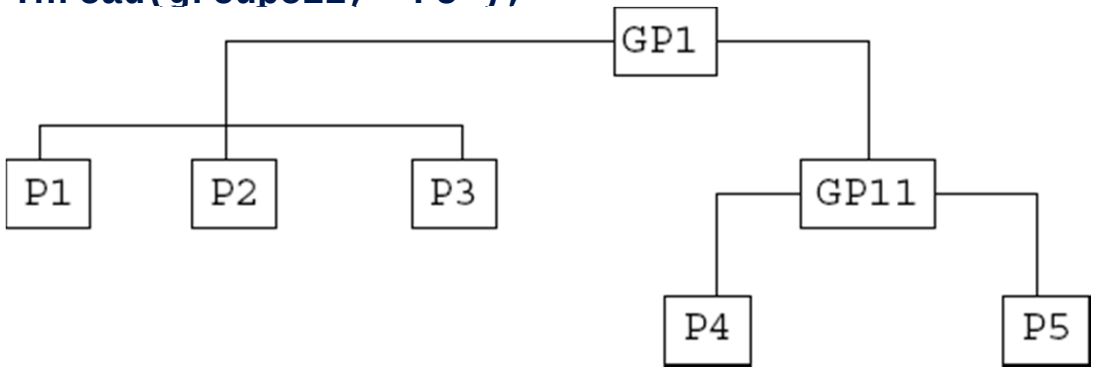
- **Arborescence :**
  - ✓ la classe **ThreadGroup** permet de constituer une arborescence de **Threads** et de **ThreadGroups**
  - ✓ elle donne des méthodes classiques de manipulation récursives d'un ensemble de threads : **suspend(), stop(), resume(),** ...
  - ✓ et des méthodes spécifiques : **setMaxPriority()**, ...
- **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 connaître son groupe
- **Création :**

```
ThreadGroup groupe = new ThreadGroup("Mon groupe");
Thread p1 = new Thread(groupe, "P1");
Thread p2 = new Thread(groupe, "P2"); .....
```

- 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

```
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