Threads et autres joyeusetés

Environnement

- Le monde n'est pas séquentiel
- Dans le monde objet, rien ne force à être séquentiel
- Parallélisme (threads)

Threads

- "Processus léger"
- Concurrence
- Ordonnancement

Concurrence

- Exécution **indépendante**
- Nouveau contexte mémoire, nouveau contexte d'exécution, partage éventuel des objets
- On peut démarrer un thread, mais c'est à lui de s'arrêter

Avantages

- Répartition des tâches sur plusieurs CPUs (en local ou en réseau)
- Séparation entre les tâches asynchrones
- Laisser du temps d'exécution aux processus "vitaux"

La classe Thread

- Encapsule un contexte
- Démarre un nouveau thread
- Appelle la méthode "run"

Concrètement

```
Thread t = new Thread();

t.start();

// continuer notre petite vie pépère
```

Résultat

rien.

La méthode "run"

- run ~ main sans argument
- Implémentée dans toutes les classes filles de Thread
- Doit se terminer d'elle même pour une raison ou une autre

CustomThread1

```
public class CustomThread1 extends Thread {
    int i;
    String name;

    public CustomThread1() {
        i = 0;
        name = "ping";
    }

    public void run() {
        while(i < 10) {
            System.out.println(name + (i++));
        }
    }
}
```

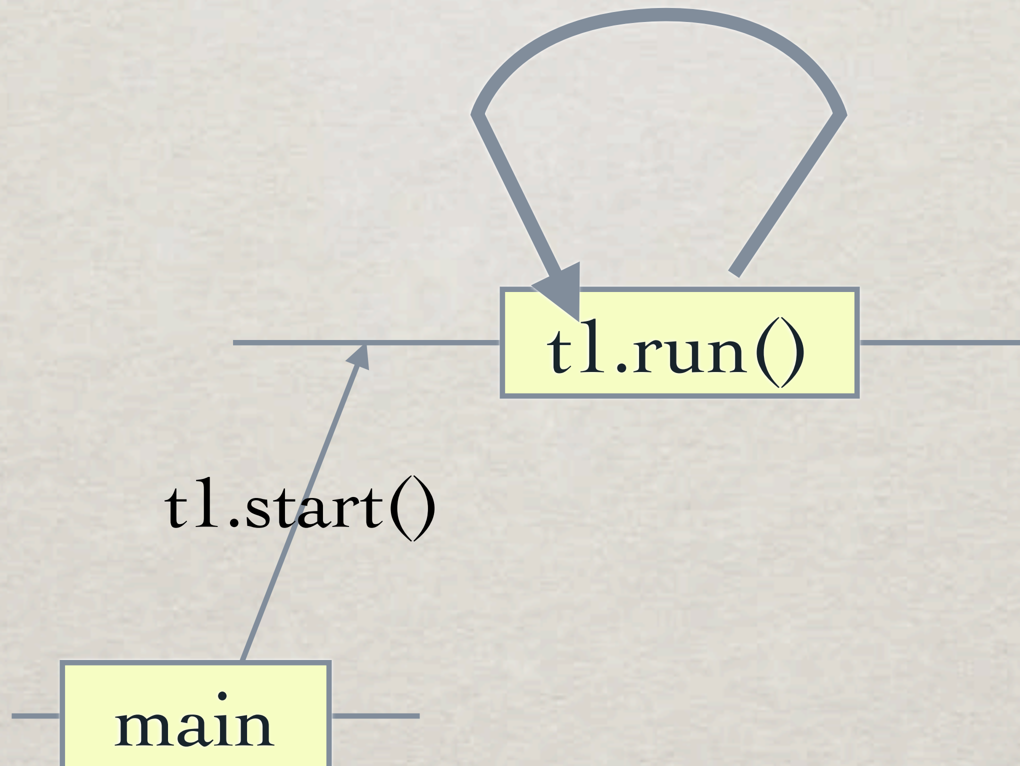
Test

```
public class TestThreads1 {
  public static void main(String [] args) {
    CustomThread1 t1 = new CustomThread1();
    t1.start();
  }
}
```

Résultat

```
ping0
ping1
ping2
ping3
ping4
ping5
ping6
ping7
ping8
ping9
```

Pourquoi?



Parenthèse

- L'application se termine si aucun thread ne reste
- System.exit(int status)

Deux threads

- Testons avec deux threads
- Pour plus de lisibilité, "ping" et "pong"
- CustomThread2

CustomThread2

```
public class CustomThread2 extends Thread {
    int i;
    String name;

    public CustomThread2() {
        i = 0;
        name = "pong";
    }

    public void run() {
        while(i < 10) {
            System.out.println(name + (i++));
        }
    }
}
```

Test

```
public class TestThreads2 {
  public static void main(String [] args) {
    CustomThread1 t1 = new CustomThread1();
    CustomThread2 t2 = new CustomThread2();
    t1.start();
    t2.start();
  }
}
```

Résultat

```
ping0
ping1
ping2
ping3
ping4
ping5
ping6
ping7
ping8
ping9
pong0
pong1
pong2
pong3
pong4
pong5
pong6
pong7
pong8
pong9
```

Ça ne marche pas!

- Exécution trop rapide, on a le temps de finir avant le second.
- Deuxième test, on augmente le nombre de pas a 10000

Résultat

```
(...)
ping8904
ping8905
pong9300
pong9301
ping8906
pong9302
ping8907
pong9303
ping8908
pong9304
ping8909
ping8910
pong9305
pong9306
pong9307
ping8911
ping8912
(...)
```

Analyse

- Les threads ne sont utiles que pour des processus **longs**
- L'ordre dans lequel ils sont exécutés est non-prévisible
- setPriority() ne marche pas vraiment

Ordonnancement

- Trouver un moyen de "bloquer" les threads en attendant un certain évènement

- Interdire l'accès concurrent aux mêmes variables (verrous)

Bloquer un thread

- Attente "active" : Thread.sleep(int millis)
- Attente "passive" : Thread.yield()

Test : modification de CustomThread*

```
// Yield = demander si d'autres n'ont pas envie
d'avancer
// Sleep = ne rien faire jusqu'a un certain
avenir

    public void run() {
        while(i < 10) {
            System.out.println(name + (i++));
            Thread.yield();
        }
    }
```

Resultat

```
ping0
pong0
ping1
pong1
ping2
pong2
ping3
pong3
ping4
pong4
ping5
pong5
ping6
pong6
ping7
pong7
ping8
pong8
ping9
pong9
```

Youpi!

- Ordonnancement "poli"
- Sleep = mise en pause
- Yield = passer la main

Files d'attente

- Problème : lire et écrire dans des variables communes
- Accès concurrent = mauvaise idée
- Verrous

Verrous

- Verrouiller un objet se dit : synchronized(objet)
- On le libère à la fin du bloc
- Exemple :

```
synchronized(this) {
  // faire quelque chose
}
```

Verrous (II)

- Tant que l'objet est verrouillé, personne ne peut le modifier ou appeler de méthode dessus

- Le verrou est transmissible : un appel de fonction dans un verrou objet garde le verrou

```
synchronized(o) {
  o.saMethode(); // o est toujours verouille
  this.maMethode(o); // o est tjrs verouille
}
```

Files d'attente

- Si deux objets tentent d'accéder a un verrou en même temps, l'un des deux l'obtient
- L'autre est mis en "file d'attente" tant que l'objet n'est pas déverrouillé

Dead Locks

- Supposons A prend le verrou v1, et essaie de prendre le verrou v2
- Supposons B prend le verrou v2, et essaie de prendre le verrou v1
- Les deux sont en file d'attente => dead lock

Astuces

- Prendre les verrous en séquence (jamais deux à la fois)
- Prendre les verrous toujours dans le même ordre (v1 **puis** v2)

Questions?