INF5171: Laboratoire #7
Programmation parallèle et concurrente en Java

9 novembre 2017
13h30–15h30
PK-S1565

Le but de ce laboratoire est de vous familiariser avec l'utilisation des constructions pour le parallélisme en Java et avec l'utilisation et la mise en oeuvre de moniteurs en Java.

---

1 Parallélisme récursif *fork/join* en Java

Obtenez une copie des fichiers pour le labo en exécutant la commande suivante sur `japet` :

```
$ git clone http://www.labunix.uqam.ca/~tremblay/git/Arbre.git
```

Vous allez devoir définir différentes méthodes sur des classes pour des arbres n-aires — donc des arbres avec un, deux ou plusieurs enfants. Ces arbres sont définis dans les classes `Arbre` (superclasse abstraite), `Noeud` (interne) et `Feuille`. Notez que les méthodes **pour les cas de base** sont déjà définies — dans les classes `Arbre` et `Feuille`. Vous devez donc uniquement définir ces méthodes **dans la classe** `Noeud`.

Pour simplifier le code, les arbres *ne sont pas génériques* — donc le champ `valeur` de chaque noeud (interne ou feuille) est toujours **un entier**.

La figure 1 à la page suivante présente un diagramme de classes UML décrivant l'organisation de ces classes et de leurs méthodes — les méthodes `equals`, `similaire` et `toString` sont omises sur le diagramme et sont déjà définies.

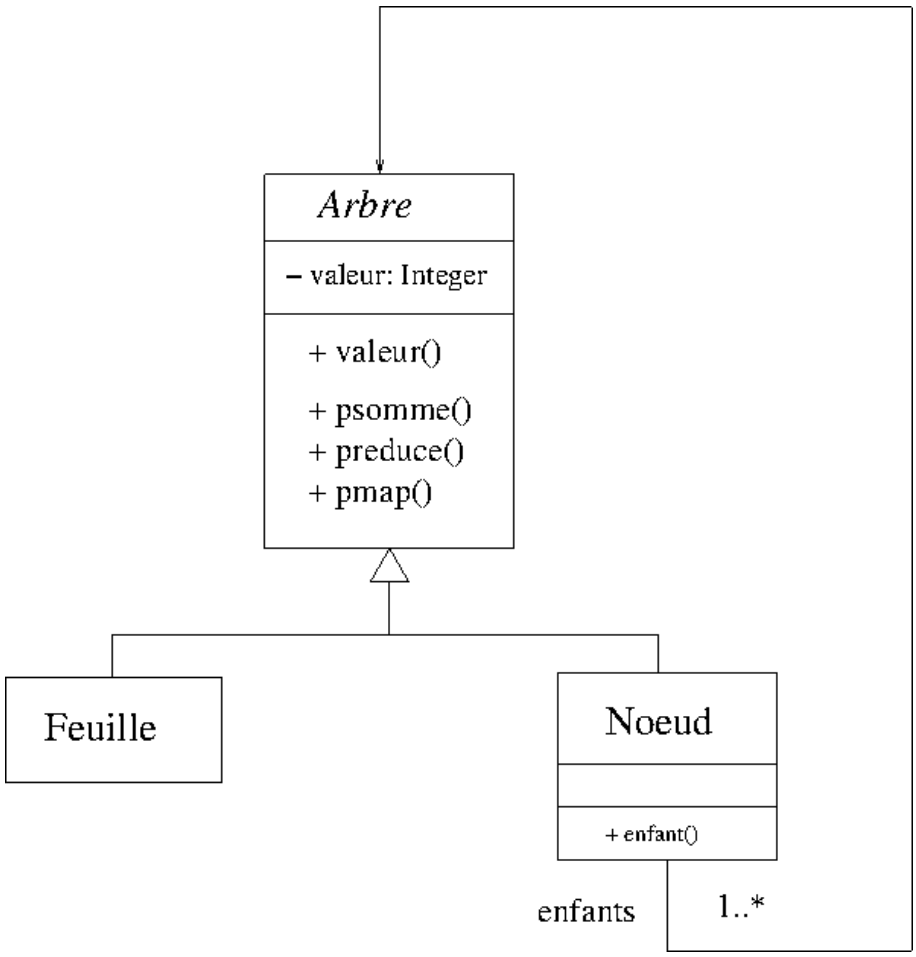

Figure 1: Diagramme de classes UML pour des arbres n-aires.

a. `psomme()` : Méthode qui fait la somme des champs `valeur` des différents noeuds de l'arbre.

   Exemple :

```
Arbre a0 = new Noeud( 12, new Feuille(10), new Feuille(20) );
Arbre a  = new Noeud( 100, a0, a0 );

assertEquals( 100 + (12 + 10 + 20) + (12 + 10 + 20),
              a.psomme() );
```

   La mise en oeuvre doit utiliser du **parallélisme récursif avec des *threads* de base explicites**. L'objectif est de comprendre le fonctionnement des *threads* de base, même si on utilise rarement une telle approche depuis l'introduction de la bibliothèque `java.util.concurrent`.

b. `preduce()` : Méthode parallèle de **réduction** par un opérateur binaire, avec parallélisme récursif **utilisant des `Futures`** avec un *pool de threads*.

   Exemple :

```
Arbre a0 = new Noeud( 12, new Feuille(10), new Feuille(20) );
Arbre a  = new Noeud( 100, a0, a0 );

assertEquals( 100 * (12 * 10 * 20) * (12 * 10 * 20),
              a.preduce(  (x, y) -> x * y )
```

c. `pmap()` : Méthode qui applique une lambda-expression sur chacun des champs `valeur` d'un arbre pour produire un nouvel arbre **avec la même structure** — voir la méthode `similaire()` — mais avec des champs valeurs obtenus par l'application de la fonction. La mise en oeuvre de la méthode doit utiliser des `Futures` et un *pool de threads* :

```
Arbre a0 = new Noeud( 12, new Feuille(10), new Feuille(20) );
Arbre a  = new Noeud( 100, a0, a0 );

Arbre r0 = new Noeud( 120, new Feuille(100), new Feuille(200) );
Arbre r  = new Noeud( 1000, r0, r0 );

assertTrue( r.equals( a.pmap( (x) -> x * 10 ) ) );
```

Suggestions et indices :

- Soit des objets satisfaisant les interfaces suivantes :

```
UnaryOperator<Integer> f
BinaryOperator<Integer> binop
```

  On appelle ces opérations — des *lambda*-expressions — avec `apply` :

```
Integer r, r0, r1, r2, ...

r = f.apply( r0 );

r2 = binop.apply( r0, r1 );
```

- L'annotation suivante, apparaissant à deux endroits, est nécessaire car ces méthodes doivent allouer un tableau d'objets génériques (des futures). Par exemple :

```
@SuppressWarnings("unchecked")
Integer psommeFuture() {
  Future<Integer>[] futures = new Future[nbEnfants];
  ...
```

- Tout appel à `join()` ou `get()` doit être fait à l'intérieur d'un `try/catch` :

```
try {
  unThread.join();
} catch( InterruptedException e ) {
  assert false : "*** Erreur ...";
}

...

try {
  unFuture.get();
} catch( ExecutionException | InterruptedException e ) {
  assert false : "*** Erreur ...";
}
```

2 Le problème des violonistes

Pour le premier exercice, vous devez obtenir le code source à compléter à l'aide de la commande suivante :

```
$ git clone http://www.labunix.uqam.ca/~tremblay/git/Violonistes.git
```

Note : Ce problème est une *variante*, plus simple, du problème des *dining philosophers*.

Le fichier `Violoniste.java` définit une classe `Violoniste` qui vise à simuler un violoniste ayant le comportement suivant :

- Lors de la création d'un `Violoniste`, on spécifie le numéro d'identification du violoniste (pour la trace d'exécution) ainsi que le nombre de pièces que ce violoniste doit jouer.
- Un violoniste alterne entre des moments où il va `dormir()` (pour une période de temps aléatoire), puis se réveiller et tenter de `jouer()` une pièce (elle aussi d'une durée aléatoire) : voir `dormir()` et `jouer()` dans le fichier `Violoniste.java`.
- Un violoniste tente tout d'abord d'obtenir un violon, donc attend qu'un violon devienne disponible si aucun ne l'est. Une fois un violon obtenu, il tente alors d'obtenir un archet. Si aucun archet n'est disponible, là aussi il attend... **mais sans restituer le violon déjà obtenu!**

Les différents violonistes doivent se partager un **nombre limité de violons et d'archets**. Pour jouer une pièce, le violoniste doit évidemment avoir en sa possession **un violon et un archet**.

Le nombre de violons et le nombre d'archets disponibles sont spécifiés sur la ligne de commande — voir plus bas — alors que l'ensemble des violons et archets disponibles à un instant sont spécifiés par des variables de classe — `violons` et `archets`.

Le fichier `Violoniste.java` contient une méthode `main` qui utilise la classe `Violoniste` en créant un certain nombre de *threads*/violonistes. Les arguments spécifiés sur la ligne de commande sont les suivants :

0. `nbViolons`
1. `nbArchets`
2. `nbViolonistes`
3. `nbPieces`

Une trace possible produite par l'exécution de ce programme est présentée à la Figure 1. Complétez la classe `Violoniste` pour obtenir le comportement désiré.

Quelques remarques

- Il est possible d'associer **plusieurs variables de condition à un même verrou**.
- L'identificateur `Lock` dénote *une interface*, alors que l'identificateur `ReentrantLock` dénote *une classe concrète* — on peut créer/instancier une classe concrète, mais pas une interface (sauf de façon indirecte avec une classe interne anonyme).
- Pendant qu'un violoniste joue, il ne doit évidemment pas rester en possession du verrou d'exclusion mutuelle du moniteur. Il ne doit tenter d'acquérir le verrou que lorsqu'il tente d'obtenir ou lorsqu'il retourne un violon ou un archet.
- Le verrou `mutexImpression` sert uniquement pour assurer **que l'impression de la trace se fait de façon atomique**. Pour assurer l'accès exclusif aux violons et archets, vous devez définir et utliser un autre verrou.
- Soit `a` un objet de type `Set`. Les opérations de base sur `a` dont vous aurez besoin sont les suivantes :
  - Pour retirer l'élément `x` de `a` : `a.remove(x);`
  - Pour ajouter l'élément `x` à `a` : `a.add(x);`
  - Pour obtenir un élément **arbitraire** de `a` : `a.iterator().next()`.

```
$ java -ea -classpath .:$(JUNIT) Violoniste 3 2 4 3
*** Les 4 violonistes vont jouer 3 pieces avec 3 violons 2 archets

        + Violoniste #4: debute  piece 1 (violon 1, archet 1)
    + Violoniste #2: debute  piece 1 (violon 2, archet 2)
    - Violoniste #2: termine piece 1 (violon 2, archet 2)
    + Violoniste #2: debute  piece 2 (violon 2, archet 2)
        - Violoniste #4: termine piece 1 (violon 1, archet 1)
      + Violoniste #3: debute  piece 1 (violon 1, archet 1)
    - Violoniste #2: termine piece 2 (violon 2, archet 2)
    + Violoniste #2: debute  piece 3 (violon 2, archet 2)
      - Violoniste #3: termine piece 1 (violon 1, archet 1)
  + Violoniste #1: debute  piece 1 (violon 3, archet 1)
  - Violoniste #1: termine piece 1 (violon 3, archet 1)
        + Violoniste #4: debute  piece 2 (violon 1, archet 1)
        - Violoniste #4: termine piece 2 (violon 1, archet 1)
    - Violoniste #2: termine piece 3 (violon 2, archet 2)
      + Violoniste #3: debute  piece 2 (violon 3, archet 1)
      - Violoniste #3: termine piece 2 (violon 3, archet 1)
  + Violoniste #1: debute  piece 2 (violon 1, archet 1)
      + Violoniste #3: debute  piece 3 (violon 2, archet 2)
      - Violoniste #3: termine piece 3 (violon 2, archet 2)
        + Violoniste #4: debute  piece 3 (violon 3, archet 2)
  - Violoniste #1: termine piece 2 (violon 1, archet 1)
        - Violoniste #4: termine piece 3 (violon 3, archet 2)
  + Violoniste #1: debute  piece 3 (violon 1, archet 1)
  - Violoniste #1: termine piece 3 (violon 1, archet 1)
```

Figure 2: Une trace possible d'exécution pour le programme `Violoniste.java`.