TME 1 : Programmation, compilation et exécution en Java

Objectifs pédagogiques :

- **classes**
- **instances**
- **tableaux**
- **itérations**

1.1 La classe `MatriceEntiere`

Le but de cet exercice est d'écrire une classe `MatriceEntiere` offrant les fonctionnalités suivantes :

- la possibilité d'initialiser les éléments de la matrice à partir de données contenues dans un fichier ASCII;
- la possibilité d'exporter dans le même format à l'aide de la méthode standard `public String toString()`;
- les opérations de base : somme de matrices, produit de matrices, produit d'une matrice par un scalaire;
- le calcul de la matrice transposée;

Question 1. Réalisez la classe MatriceEntiere qui est munie d'un tableau à deux dimensions d'entiers. Son constructeur prend en paramètre le nombre de lignes et le nombre de colonnes de la matrice et initialise les éléments à zéro. Ajoutez aussi les accesseurs suivants : `public int getElem(int lig, int col)` et `public void setElem(int lig, int col, int val)` ainsi que les méthodes `public int nbLignes()` et `public int nbColonnes()`.

NB: Les tableaux Java ont un attribut `length` qui donne la taille du tableau.

Question 2. Ajouter l'opération `public static MatriceEntiere parseMatrix(File fichier) throws IOException`. Cette opération doit permettre de lire les fichiers de données fournis et de construire la matrice correspondante. Référez-vous aux tests fournis pour le format précis.

Question 3. Ajoutez à la classe `MatriceEntiere` une méthode `public String toString()` qui produit une répresentation sous la forme de String de la matrice, dans un format compatible avec les fichiers de test fournis.

Question 4. Ajoutez également une comparaison d'égalité logique standard entre deux matrices `public boolean equals(Object o)`.

Question 5. Ajoutez une méthode `public MatriceEntiere ajoute(MatriceEntiere m)throws TaillesNonConcordantesE` qui retourne la somme de la matrice courante et de m, où lève une exception (que vous définirez) si les dimensions ne sont pas concordantes.

Question 6. On rappelle que le produit des matrices $A(l_A, c_A)$ et $B(l_B, c_B)$ peut être calculé si $c_A = l_B$. Le résultat est une matrice $P(l_A, c_B)$ dont l'élément en position (i, j) a pour valeur :

$$p_{i,j} = \sum_{k=0}^{c_A - 1} a_{i,k}.b_{k,j}$$

Ajoutez `public MatriceEntiere produit(MatriceEntiere m)throws TaillesNonConcordantesException` qui calcule ce produit (ou lève une exception si les dimensions ne sont pas concordantes).

Question 7. Ajouter la méthode `public MatriceEntiere transposee()` qui retourne la transposée de la matrice.

1.2 String vs StringBuilder

La String de Java est immuable; pour cette raison il est toujours recommandé quand on construit une grande string petit à petit d'utiliser la classe StringBuilder qui est mutable. Cet exercice vous permettra d'apprécier la différence en pratique.

Question 8. On vous fournit une classe Repeat et un test associé. La classe actuellement ne passe pas le test, corrigez l'implantation pour utiliser un StringBuilder.

Question 9. Avant de finir la session pensez bien à push votre travail sur le dépôt.

TME 2 : Parallélisme, Thread

Objectifs pédagogiques :

- **Thread**
- **Runnable**
- **start, join**

2.1 Compter les mots

On fournit un programme WordCount capable de compter le nombre de mots dans un ou plusieurs fichiers.

WordCount

```java
package pc.countwords;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Arrays;

public class WordCount {

   public static int countWords(String filename) throws IOException {
      long startTime = System.currentTimeMillis();
      try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
         int total = 0;
         for (String line = br.readLine(); line != null; line = br.readLine()) {
            total += line.split("\\s+").length;
         }
         System.out.println("Time for file "+filename+" : "+(System.currentTimeMillis()-
             startTime) + " ms for "+ total + " words");
         return total;
      }
   }

   public static void main(String[] args) {
      long startTime = System.currentTimeMillis();
      int [] wordCount = new int[args.length];
      for (int i = 0; i < args.length; i++) {
         try {
            wordCount[i] = countWords(args[i]);
         } catch (IOException e) {
            System.err.println("Error reading file: " + args[i]);
            e.printStackTrace();
         }
      }
      System.out.println("Word count:" + Arrays.toString(wordCount));
      int total = 0;
      for (int count : wordCount) {
         total += count;
      }
      System.out.println("Total word count:" + total);
      System.out.println("Total time "+(System.currentTimeMillis()-startTime) + " ms");
   }
}
```

Pour l'exécuter, on recommande de lancer la commande suivante dans le dossier racine du projet (configurer les arguments du bouton “Play" de votre IDE est possible aussi mais plus pénible) :

```
cd ~/git/ProgrammationConcurrente-LU3IN001/tme2
java -cp bin/ pc.countwords.WordCount ~/*/*
```

Choisissez un dossier contenant beaucoup de fichiers pour tester le programme et que l'éxecution dure au moins quelques secondes.

Question 1. Modifiez le programme pour qu'il utilise un Thread différent pour chaque fichier à traiter. Chaque Thread devra afficher le nombre de mots dans le fichier qu'il traite et copier cette valeur dans le tableau "wordCount". On pourra introduire une classe dérivant de Thread ou qui implémente Runnable au choix.

2.2 Matrices en parallèle

On reprend la classe MatriceEntiere du TME1. On recommande de la recopier dans le dossier `src/pc` du TME2.

Question 2. Implanter une méthode `public MatriceEntiere produitParScalaireMT(int n)` qui calcule le produit scalaire en utilisant un thread par ligne du résultat.

Question 3. Dans le même esprit, implanter une méthode `public MatriceEntiere produitMT(MatriceEntiere m)throws TaillesNonConcordantesException` qui calcule le produit de deux matrices en utilisant un thread par cellule de la matrice résultat.

NB: on peut remarquer que dans ces deux fonctions, les arguments sont accédés en lecture seule, donc il est inutile de les protéger des accès concurrents. De même chaque thread écrit dans une case différente du tableau résultat, donc il n'y a pas de conflit d'écriture.

NB: on recommande de bien réfléchir à la manière de découper le travail en tâches pour les threads, et sur les données dont ils ont besoin pour travailler et déposer leur résultat. On pourra traiter une question avec une classe qui étend Thread et une autre avec une classe qui implémente Runnable pour varier un peu les syntaxes utilisées.

Question 4. Ajouter un test qui contrôle que le résultat des opérations classiques et des versions MT sont concordantes, et qui affiche les temps pris par chaque approche.

Question 5. N'oubliez pas de push votre travail à la fin de la séance.

TME 3 : Verrouillage Fin

Objectifs pédagogiques :

- **class MT safe**
- **verrouillage fin**
- **contention**

3.1 Introduction

L'objectif de ce TME est de revisiter la classe représentant une liste chaînée du TD1 pour la rendre thread-safe.

On vous fournit deux version de la liste, une itérative et un récursive.

3.2 Une liste chaînée thread-safe

Question 1. En vous basant sur la classe `pc.iter.SimpleList` fournie, écrivez une version thread-safe de la liste chaînée `pc.iter/SimpleListSync` en manipulant simplement le moniteur de l'objet principal qui héberge `head`.

Question 2. En vous basant sur la classe `pc.rec.SimpleListRec` fournie, écrivez une version thread-safe de la liste chaînée `pc.rec/SimpleListRecSync` en manipulant simplement le moniteur de l'objet principal qui héberge `head`.

Question 3. Ecrire un test qui prend en argument une occurrence de l'interface `IList`, et crée N threads qui ajoutent chacun M éléments à la liste. Ajoutez aussi N threads qui invoquent `contains` M fois chacun avec des éléments qui n'existent pas (on placera un assert pour contrôler que c'est le cas). Enfin, une fois que tous ces threads sont terminés, vérifier que la taille de la liste est bien $N \times M$. Mesurez et affichez le temps total pris pour le test. Testez les 4 implémentations de la liste. On pourra significativement augmenter la valeur de M pour augmenter la contention. On recommande de définir des classes déclarées avec le mot clé "static" directement au sein de la classe de test pour héberger les méthodes run nécessaires. On recommande de construire une fonction `void runConcurrentTest(IList<String> list, int N, int M)` pour faire facilement varier l'implantation de IList choisie ainsi que les paramètres M et N.

Question 4. Ajouter d'autres tests qui sollicitent les autres opérations de la liste sous une situation de forte contention.

3.3 Un verrouillage par chainon

Dans cette partie, on souhaite utiliser un verrouillage plus fin, qui permet à plusieurs thread potentiellement d'accéder à la liste en même temps. Le principe est de verrouiller uniquement le chaînon auquel on accède.

On prendra garde à :

- L'accès à "head" doit être protégé ; on se synchronisera sur l'objet principal (this) pour cela. Une fois la valeur accédée, on peut relâcher le verrou.
- L'accès à next doit être protégé en se synchronisant sur le chaînon qui l'héberge.
- Une fois le champ next accédé, on peut relâcher le verrou sur le chaînon courant avant de poursuivre le traitementsur le chaînon suivant.

Question 5. En vous basant sur la classe `pc.iter.SimpleList` fournie, écrivez une version thread-safe de la liste chaînée `pc.iter/SimpleListFine` qui utilise une stratégie de verrouillage par chainon.

Pour vous guider, voici une version possible de la méthode `contains` :

`SimpleListFine`

```
@Override
public boolean contains(T element) {
   Chainon<T> cur;

   synchronized (this) {
      cur = head;
   }
   while (cur != null) {
      synchronized (cur) {
         if (cur.data.equals(element)) {
            return true;
         }
         cur = cur.next;
      }
   }
   return false;
}
```

Question 6. En vous basant sur la classe `pc.rec.SimpleListRec` fournie, écrivez une version thread-safe de la liste chaînée `pc.iter/SimpleListRecFine` qui utilise une stratégie de verrouillage par chainon.

Pour vous guider, voici une version possible de la méthode `contains` :

Dans Chainon

```
public boolean contains(T element) {
   synchronized (this) {
      if (data.equals(element)) {
         return true;
      } else if (next == null) {
         return false;
      }
   }
   return next.contains(element);
}
```

Dans Liste

```
public boolean contains(T element) {
   Chainon<T> cur;
   synchronized (this) {
      if (head == null) {
         return false;
      } else {
         cur = head;
      }
   }
   return cur.contains(element);
}
```

3.4 Conversion en String

Cette question est considérée comme un bonus, à traiter si vous avez terminé les questions précédentes.

Question 7. Ajoutez une méthode `public String toString()` aux implantations séquentielles de Liste fournies. Vous réaliserez une version itérative et une version récursive. Attention à éviter une approche qui cumule le résultat directement dans un String, et donc utilisez plutôt un StringBuilder.

TME 4 : Le diner des Philosophes

Objectifs pédagogiques :

- deadlock
- ressources partagées

4.1 Le problème du dîner des philosophes

Nous souhaitons programmer un problème classique de programmation concurrente, “le dîner des philosophes", dans lequel `N` philosophes sont assis autour d'une table ronde. Il y a `N` baguettes sur la table, chaque baguette est posée entre deux philosophes. Pour un dîner de 5 philosophes, la configuration est la suivante :

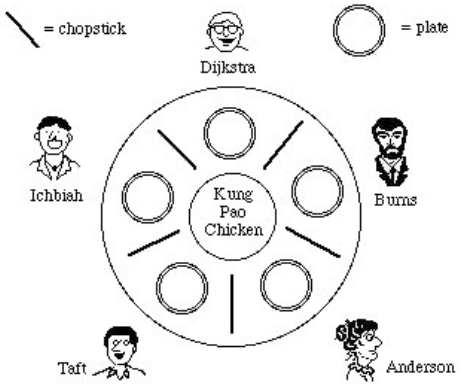

Figure 1: Le dîner des 5 philosophes

Lorsqu'il est fatigué de penser, un philosophe doit manger pour reprendre des forces. Pour cela, il faut qu'il s'empare de la baguette qui se trouve à sa gauche et de celle qui se trouve à sa droite. Il peut manger dès qu'il y parvient. Une fois rassasié, le philosophe repose ses baguettes et repart dans une phase de réflexion.

Question 1. Programmez la classe `Fork` qui décrit le comportement d'une baguette. Elle dispose de deux méthodes `public void acquire()` et `public void release()`. Si une baguette n'est pas disponible, la méthode qui permet de prendre la baguette doit mettre le `thread` appelant en attente. Libérer la baguette permet de réveiller un philosophe en attente de cette baguette. On s'appuiera sur une instance de `ReentrantLock` pour réaliser la sémantique adaptée.

Question 2. Programmez la classe `Philosopher` qui décrit le comportement d'un philosophe. C'est une classe active, qui implémente `Runnable`. On lui passe à la construction les baguettes qui lui sont associées. Il réalise une boucle infinie où il pense, acquiert sa baguette de gauche, puis celle de droite, mange, et enfin relache ses baguettes. On s'assurera de bien identifier la progression du philosophe en affichant un message au moins quand le philosophe “pense”, “possède une baguette” et “mange”. On pourrait aussi ajouter des délais aléatoires, mais ça diminue la contention ce qui rendra plus difficile d'exhiber des fautes de concurrence.

Question 3. Programmez une classe `TestPhilo` qui permet de tester les deux classes que vous venez de programmer. On instanciera un tableau ou une List de Fork, et autant de Philosopher que de Fork. On les lancera tous en parallèle.

Question 4. Quel problème peut-on rencontrer à l'exécution de ce programme ?

Question 5. Si l'on suppose une numérotation naturelle des baguettes de 0 à N-1, quel philosophe ne respecte pas l'ordre naturel d'acquisition des baguettes ? Corrigez le problème en reconfigurant ce philosophe à la construction.

4.2 Terminaison du programme

Les philosophes actuellement ne s'arrêtent jamais. On souhaite au contraire pouvoir terminer le programme proprement. Le programme principal doit maintenant dormir quelques secondes après avoir crée et lancé les philosophes, puis les interrompre tous et attendre leur terminaison avant de terminer le programme.

Question 6. Modifiez le programme principal pour obtenir ce comportement. On se contentera d'invoquer "interrupt" sur chaque philosophe pour les arrêter. En conséquence adaptez le code des philosophes pour qu'ils réagissent correctement à une interruption en testant à chaque tour de boucle s'il a été interrompu.

Question 7. Cette approche fonctionne-t-elle sur la version avec deadlock ? Pourquoi ? Lisez attentivement la documentation de la méthode "lock()" `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/locks/Lock.html` et chercher une façon d'acquérir la baguette qui permette de réagir à une interruption et donc de terminer le programme.

Question 8. Pensez à push votre travail sur votre dépôt git.