TP de Compléments en Programmation Orientée Objet n° 11 : Problèmes récursifs avec `ForkJoinPool`, `RecursiveAction`,

Exercice 1 :

Faites tout d'abord l'exercice 3 du TP 10 pour assimiler les `wait`/`notify`

Pour les exercices suivants, on rappelle que :

— `invoke` et `invokeAll` permettent de lancer un ou plusieurs processus, qui sont réputés être terminé au moment où on exécute les instructions qui suivent ces appels.

— pour qu'un objet `RecursiveAction` apporte le résultat d'un calcul, la technique consiste à lui ajouter un attribut `resultat` qu'on pourra récupérer en prévoyant de fournir une methode `getResultat()`

Exercice 2 :

En reprenant la syntaxe des exemples vu cours, définissez une classe étendant `RecursiveAction` construite autour d'un entier. Elle lancera 2 nouvelles instances, construites autour de la moitié de cet entier et ainsi de suite tant que l'entier est positif. A chaque scission affichez un symbole + et testez votre exemple sur un `ForkJointPool`. Cet exercice ne calcule rien, mais permet de revoir la syntaxe ensemble.

Modifiez votre travail pour qu'à la fin on puisse afficher le nombre de '+' visibles, obtenu par une méthode `getResultat()`

Exercice 3 : Tri fusion pour les entiers

On vous donne ici le code d'une implémentation du tri fusion que vous connaissez déjà bien. En suivant la même démarche que dans l'exercice 1, proposez en une version parallélisée.

```
1
2
3     public static int[] triInt(int [] tab) {
4         if (tab.length <= 1) return tab;
5         else {
6             int pivot = Math.floorDiv(tab.length, 2);
7             int[] t1 = Arrays.copyOfRange(tab, 0, pivot);
8             int[] t2 = Arrays.copyOfRange(tab, pivot, tab.length);
9             t1=triInt(t1);
10            t2=triInt(t2);
11
12            return fusionInt(t1, t2);
13        }
14    }
15
16    public static int[] fusionInt(int[] t1, int[] t2) {
17       int i1=0, l1=t1.length,i2=0, l2=t2.length;
18       int[] t3= new int[l1+ l2];
19       int i3=0;
20
21        while (i1 < l1 ||i2 < l2)
22            if (i1 < l1 && ( !(i2 < l2) || t1[i1] < t2[i2] ))
23                t3[i3++] = t1[i1++];
24            else  t3[i3++] = t2[i2++];
25        return t3;
26    }
```

```
27
28          public static void main(String[] args) {
29              int[]tab= {8, 4, 7, 1, 2, 9, 4, 3, 5, 7};
30              int []res = TriFusion.triInt(tab);
31
32              for (int x:res) System.out.print (x+" ");
33              System.out.println();
34
35  }
```

Exercice 4 :

On va généraliser ce qu'on vient de faire dans 2 directions :

— sur les données : en remplaçant `int` par `Integer` et `[]` par `List` l'idée étant de manipuler des objets plutôt que des types de base [1]. On en profitera aussi pour présenter une autre façon d'obtenir le résultat que calcule une tache.

— sur le parallélisme : en prenant un peu plus de liberté. On rappelle que `invoke` et `invokeAll` synchronisent assez fortement les taches puisqu'elles ne redonnent la main au processus courant que lorsque les taches invoquées sont terminées. On montrera qu'on peut expliciter un peu plus, ce qui vous donnera d'autres options pour résoudre des problèmes.

1. Reprenez votre code, et faites le remplacement des `int` par des `Integer` et des `[]` par des `List` et `ArrayList`

2. Vous avez en mémoire que c'est dans `invokeAll` que le branchement qui définit de nouveaux processus à exécuter en parallèle ce faisait, et qu'après cette méthode on ne revient à la suite courante des exécutions que quand les actions lancées ont été terminées.

   D'un autre coté, pour qu'une `RecursiveAction` produise un résultat nous avons fait un travail d'internalisation de cette donnée et nous avons également implémenté son accesseur.

   Il y a une alternative : elle consiste d'abord à décomposer ce qui se passe dans `invoqueAll`, en le remplaçant par `tache1.fork()` et `tache2.fork()` : `fork` est une commande qui lance une tache tout en laissant le processus courant continer sa vie.

   Cela nous permet à présent de choisir un type de tache mieux adaptée au cas qui nous intéresse : une tache qui retourne un résultat naturellement, sans qu'on doive l'internaliser un peu artificiellement.

   `RecursiveTask<E>` est justement prévue pour produire un résultat de type `E`. On peut donc lancer le calcul avec `fork` faire d'autres choses, et pour obtenir le résultat on appellera la méthode `join`, c'est elle qui retourne la valeur attendue. (On comprend qu'au besoin elle attendra que la tache termine pour que le résultat soit connu)

   Reprenez votre code en remplaçant le couple `invokeAll`/ `RecursiveAction` par `fork-join`/`RecursiveTask<List<Integer>>`

3. Le dernier travail intéressant que l'on peut faire sur cet exercice ne concerne plus le parallélisme, il est donc optionnel.

   Le tri fusion peut être généralisé en remplaçant les entiers par un type générique `<E>` d'éléments comparables entre eux.

   L'interface `Comparable<T>` requiert de définir `int compareTo(T o)` et permet de s'assurer qu'on peut désigner un plus petit ou un plus grand parmi les éléments de type T.

1. à la toute fin de l'exercice on rendra même générique le type qu'on souhaite trier

On pourrait dans un premier temps penser à `E extends Comparable<E>`, mais malheureusement il n'est pas satisfaisant pour la raison suivante : vous pouvez imaginer une situation où `B` hérite d'une classe `A` qui implémente `Comparable<A>`, mais qu'on n'ait pas prévu que `B` implémente précisément `Comparable<B>`.

Alors, pour trier une liste de `B`, on pourra utiliser les comparaisons prévue dans une classe supérieure, mais le type `B` sera refusé par le filtre `E extends Comparable<E>` ...

Le type générique qui convient est assez dur à trouver, le voici :

```
E extends Comparable<? super E>
```

- Faites cette dernière amélioration à votre code, et rendez générique la spécialisation initialement faite sur les `Integer`