Correction examen Java Avancé Ratrappage 2016

M1 MIAGE apprentissage

—2h. Uniquement la javadoc est autorisée.—

▸ Exercice 1. Échangeur

On veut pouvoir faire un échange de valeurs entre 2 threads. Pour cela, on va créer une classe `Exchanger`. Le thread 1 va envoyer une valeur v_1 à l'échangeur au moyen d'une méthode `exchange`. L'échangeur va bloquer et faire attendre le thread 1 jusqu'à ce que le thread 2 fasse également appel à `exchange` avec sa propre valeur v_2. À ce moment là, `exchange` renvoi v_1 au thread 2 et débloque le thread 1, dont l'appel à `exchange` renvoi v_2.

En fait, cette classe existe déjà dans la JDK, on va donc la ré-implémenter. Voici un exemple d'utilisation :

```
public static void main(String[] args) throws InterruptedException {
    Exchanger<String> exchanger = new Exchanger<>();
    new Thread(new Runnable() {
      @Override public void run() {
        try {
          System.out.println("thread 1 " + exchanger.exchange("miage")); //affiche "thread 1 null"
        } catch (InterruptedException e) {
          throw new AssertionError(e);
        }
      }
    }).start();
    System.out.println("thread main " + exchanger.exchange(null)); //affiche "thread main miage"
}
```

1. Proposer une classe `Exchanger` avec sa méthode `exchange` **en utilisant uniquement des** `Lock` (c'est-à-dire sans bloc synchronized).
2. On veut maintenant pouvoir appeler plus de deux fois la méthode d'échange. Par exemple, le code suivant :

```
public static void main(String[] args) throws InterruptedException {
    Exchanger<String> exchanger = new Exchanger<>();
    for(int i=0 ; i<10 ; i++) {
      final int id = i;
      new Thread(new Runnable() {
        @Override public void run() {
          try {
            System.out.println("thread "+id + " received from " + exchanger.exchangeMult("thread "+id))
                ;
          } catch (InterruptedException e) { }
        }
      }).start();
    }
}
```

peut donner ce résultat :

```
thread 0 received from thread 1
thread 4 received from thread 2
thread 2 received from thread 4
thread 1 received from thread 0
thread 3 received from thread 5
thread 5 received from thread 3
thread 8 received from thread 6
thread 9 received from thread 7
thread 6 received from thread 8
thread 7 received from thread 9
```

Donner le code de `exchangeMult`, toujours en utilisant uniquement des `Lock`.

✂ -

1. Point pour le lock/condition (1), pour le wait/notify correctement sur le monitor (1), pour le while sur la valeur (0.5). Point pour le type paramétré (0.5). Pour le fonctionnement (0.5) throws interrupt dans exchange (0.5) Juste remettre le boolean.. (1) (pour plusieurs fois)

```java
public class MyExchangerLock<E> {

  private E prev = null;
  private boolean first = true;
  private final ReentrantLock lock = new ReentrantLock();
  private final Condition empty = lock.newCondition();

  public E exchange(E e) throws InterruptedException {
    lock.lock();
    try {
      if(first) {
        first = false;
        prev = e;
        while(prev == e) //utile ?
          empty.await();
        return prev;
      }
      else { //(prev != null) {
        E sav = prev;
        prev = e;
        empty.signal();
        return sav;
      }
    }
    finally {
      lock.unlock();
    }
  }
```

```java
public class ExchangerMultipleLock<V> {
  private static final int FREE=0;
  private static final int ONEVAL=1;
  private static final int BUSY=2;

  private int state;
  private V val1;
```

```java
  private V val2;
  private final ReentrantLock lock = new ReentrantLock();
  private final Condition notBusy = lock.newCondition();
  private final Condition secondVal = lock.newCondition();

  V exchange(V val) throws InterruptedException {
    Objects.requireNonNull(val);
    lock.lock();
    try {
      while(state==BUSY)
        notBusy.await();
      if (state==FREE){
        val1=val;
        state=ONEVAL;
        notBusy.signal();
        while(state==ONEVAL)
          secondVal.await();
        state=FREE;
        notBusy.signal();
        return val2;
      }
      if (state==ONEVAL){
        state=BUSY;
        val2=val;
        secondVal.signal();
        return val1;
      }
      throw new AssertionError();
    } finally {
      lock.unlock();
    }
  }
```

▸ Exercice 2. Sac

Le but est d'implémenter un `Bag`, une structure de données qui ressemble à un `Set`, mais qui en plus compte le nombre de fois où un élément est stocké. Ainsi, lorsque l'on désire insérer un élément dans un `Bag`, si cet élément est déjà présent, on incrémente le compteur du nombre d'occurrences de cet élément (sinon on l'ajoute avec nombre d'occurence 1).

On impose l'utilisation de l'interface suivante :

```java
public interface Bag<E> {
    public int add(E element);

    public int count(E element);

    public int remove(E element);

    public int addWithCount(E element, int n);

    //à compléter au fur et àmesur des questions
}
```

1. Écrire une classe `Bags`, qui contient une méthode `static createSimpleBag` qui crée et retourne un `Bag` vide. Pour cela, on utilisera une *classe interne* qu'on

appelera `BagImpl` qui implémente l'interface `Bag` (c'est ce qui est demandé pour avoir les points, mais si vous n'y arrivez pas, faites d'une autre manière afin de pouvoir faire les questions suivantes).

2. Écrire la méthode d'ajout d'un élément. Le nombre d'opérations nécessaire pour l'ajout doit être constant (i.e. ne pas dépendre de la taille du Bag). On ne veut pas pouvoir ajouter `null` dans le Bag. La méthode retourne le nombre d'éléments contenu dans le Bag (y compris le nouvel élément). Si l'élément n'était pas présent auparavant, la méthode renvoie donc 1.
3. Écrire la méthode `count` qui donne le nombre d'occurrences d'un élément (0 si l'élément est absent). Le nombre d'opérations nécessaire pour calculer cette valeur doit également être constant.
4. Écrire la méthode `remove` qui retire une occurrence de l'élément (s'il apparaissait n fois, il apparait maintenant $n-1$ fois). S'il n'y avait qu'une occurrence, l'objet est supprimé du Bag. La méthode renvoie le nouveau nombre d'occurrence de l'objet après l'appel (ou 0 si l'objet n'est pas dans le Bag).
5. Écrire la méthode `addWithCount` qui ajoute n occurrences (n strictement supérieur à 0) de l'élément en une fois. Le résultat final doit être le même qu'appeler n fois la méthode `add`. Cependant, la méthode addWithCount doit utiliser un nombre constant d'opérations. La méthode renvoie également le nombre d'occurrences de l'objet après l'appel à la méthode.
6. Ajouter une méthode `iterator` qui renvoie un itérateur sur le Bag. Si un élément est présent x fois, l'itérateur doit renvoyer x fois le même élément. On n'impose rien sur l'ordre dans lequel les éléments sont renvoyés. On veut que la méthode `remove` de l'itérateur soit implémentée (question difficile, n'y restez pas bloqué trop longtemps).
7. Faire en sorte qu'il soit possible d'utiliser la syntaxe for-each sur votre Bag.
8. Ajouter une méthode `static createOrderedByInsertionBag` dans `Bags` qui permet de créer un Bag dont les éléments sont ordonnés par rapport à l'ordre d'insertion de la première occurrence de l'élément. Par exemple, si on ajoute successivement $1, 2, 1, 1, 2, 4, 3, 4$, l'itérateur devra renvoyer $1, 1, 1, 2, 2, 4, 4, 3$. Le nombre de lignes ajoutées dans le code doit être très petit (i.e. il ne faut pas réécrire toutes les méthodes).
9. Écrire une méthode `addAll` qui prend en argument un Bag et ajoute un bag dans un bag. On veut garder l'ordre d'insertion si le bag a été créé avec `createOrderedByInsertionBag`.
10. Ajouter une méthode `asSetOfList` qui renvoie un `Set` de `List` d'éléments. Il y aura autant de listes dans le set que d'éléments différents dans le Bag. Chaque liste contiendra n fois l'élément, si l'élément était présent n fois dans le Bag. Par exemple, si dans le bag il y avait 2 fois l'élément "miage" et 1 fois l'élément "java", la méthode retourne un Set dans 2 listes, avec dans la première 2 fois "miage" et dans la seconde liste 1 fois "java". On veut que le Set et les listes qui y sont stockées soient **non-mutable**. De plus, l'ensemble doit agir comme **une vue**,

c'est-à-dire qu'une modification du Bag postérieure à l'appel de `asSetOfList` devra être visible dans ce qu'a retourné la méthode.

Vous pouvez tester votre implémentation à l'aide des tests unitaires fournis.

✂ ----------

```java
package exam.ratrap.seize;

import java.util.AbstractList;
import java.util.AbstractSet;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.Objects;
import java.util.Set;

public class Bags {

  private Bags() {
    //no
  }

  class BagImpl<E> implements Bag<E> {
    private final Map<E, Integer> map;

    private BagImpl(Map<E, Integer> map) {
      this.map = map;
    }


    @Override
    public int add(E element) {
      Objects.requireNonNull(element);
      if(!map.containsKey(element)) {
        map.put(element, 0);
      }
      map.put(element, map.get(element)+1);
      return map.get(element);
    }

    @Override
    public int count(Object element) {
      Objects.requireNonNull(element);
      if(!map.containsKey(element)) return 0;
      return map.get(element);
    }

    @Override
    public int remove(E element) {
      Objects.requireNonNull(element);
      if(!map.containsKey(element)) return 0;
      map.put(element, map.get(element)-1);
      if (map.get(element)==0) {
        map.remove(element);
        return 0;
      }
      return map.get(element);
    }

    @Override
    public int addWithCount(E element, int n) {
      if(n<1) throw new IllegalArgumentException(n + " must be positive");
```

```
59      Objects.requireNonNull(element);
60      if(!map.containsKey(element)) {
61        map.put(element, 0);
62      }
63      map.put(element, map.get(element)+n);
64      return map.get(element);
65
66    }
67
68    @Override
69    public Iterator<E> iterator() {
70      return new Iterator<E>() {
71        Iterator<E> iter = map.keySet().iterator();
72        int i = 0;
73        //E cur = iter.next();
74        E cur = null;
75        boolean canRemove = false;
76
77        @Override
78        public boolean hasNext() {
79          return (iter.hasNext() || i < map.get(cur));
80        }
81
82        @Override
83        public E next() {
84          canRemove = true;
85          if(cur == null || i>= map.get(cur)) {
86            cur = iter.next();
87            i=1;
88            return cur;
89          }
90          else {
91            i++;
92            return cur;
93
94          }
95        }
96
97        @Override
98        public void remove() {
99          if(cur == null || !canRemove) throw new IllegalStateException();
100         int curV = map.get(cur);
101         if(curV > 1) {
102           map.put(cur, curV-1);
103           i--;     //remains some to iterate for cur, we do not consider one
104         }
105         else {
106           iter.remove();
107           cur=null; //to force iter.next on the next()
108         }
109         canRemove = false;
110       }
111     };
112   }
113
114
115   @Override
116   public Set<List<E>> asSetOfList() {
117     return new AbstractSet<List<E>>() {
118
119       @Override
120       public Iterator<List<E>> iterator() {
121         return new Iterator<List<E>>() {
122           Iterator<E> it = map.keySet().iterator();
123           @Override
124           public boolean hasNext() {
125             return it.hasNext();
```

```
126           }
127
128
129           @Override
130           public List<E> next() {
131             E e = it.next();
132             return new AbstractList<E>() {
133               @Override
134               public E get(int index) {
135                 return e;
136               }
137
138               @Override
139               public int size() {
140                 return map.get(e);
141               }
142             };
143           }
144         };
145       }
146
147       @Override
148       public int size() {
149         return map.size();
150       }
151     };
152   }
153
154
155   @Override
156   public void addAll(Bag<? extends E> b) {
157     for(E e : b) {
158       add(e);
159     }
160   }
161 }
162
163 public static <E> Bag<E> createSimpleBag() {
164   return new Bags().new BagImpl<E>(new HashMap<>());
165 }
166
167 public static <E> Bag<E> createOrderedByInsertionBag() {
168   return new Bags().new BagImpl<E>(new LinkedHashMap<>());
169 }
170
171 public static void main(String[] args) {
172   Bag<String> b = Bags.createOrderedByInsertionBag();
173   b.add("lucie");
174   b.add("lucie");
175   b.add("florian");
176   b.add("denis");
177   b.add("florian");
178   b.add("florian");
179   b.add("florian");
180
181
182   for(String s : b) {
183     System.out.println(s);
184   }
185 }
186
187
188 }
```

1. 2 (classe interne + new ok)
2. 1.5 (hashmap put) plus ok put

3. 1 get hashmap
4. 1.5 (remove ok)
5. 1 (comme add en fait)
6. 2.5 pour hashnext next ok + 1.5 pour remoev
7. 1 iterable
8. 1 linkedhashmap + 1 si intelligent
9. 1 foreach + 1 extends
10. 2 abstract set + 1.5 si abstractlist dans l'iterateur

19.5...