**Cours de Programmation orientée objet** **L3 informatique**

**Examen de première session** **Année 2013-2014**

Durée 2 heures

Tous les documents sont interdits, y compris les téléphones portables et autres gadgets électroniques

Sur votre copie en plus de votre nom et prénom, veuillez indiquer votre formation : L3 info, EIDD, M1 linguistique-info, ...

*Le barème est donné à titre indicatif.*

Un *Sac* est un multi-ensemble, c'est-à-dire un ensemble d'éléments dans lequel un même élément peut être présent plusieurs fois et où l'ordre des éléments n'a pas d'importance. Par exemple, $\{1; 2; 1; 1; 0\}$ est un sac où l'élément 1 apparaît 3 fois, il est égal au sac $\{0; 1; 1; 1; 2\}$ (ordre des éléments indifférent.)

1. Définissez une interface `Sac` (paramétrée par un type `E`) qui représente un sac d'éléments de type `E` et qui contiendra
   - une méthode `ajouter(...)` qui ajoute un élément à l'ensemble et ne retourne rien ;
   - une méthode `combien(...)` qui prend un élément en argument et retourne le nombre d'éléments égaux à cet élément (deux éléments sont égaux si comparés avec la méthode `equals` elle retourne `true`.) ;
   - une méthode `retirerUn(...)` qui prend un élément en argument et en retire un exemplaire. Retourne `true` si un exemplaire a été trouvé, `false` sinon.

   (1pt).

2. Définissez une implémentation `SacTriche` (paramétrée par un type `E`) qui implémente l'interface `Sac` précédente à l'aide d'une liste chaînée `LinkedList`. (Un rappel de quelques fonctions de `LinkedList` est donné en annexe.) (2pts)

3. On veut définir une implémentation `SacTriche<Integer>` qui implémente l'interface `Sac<Integer>` à l'aide d'un tableau d'`Integer` de taille fixe.

   Les éléments du sac correspondent aux cases ne contenant pas `null` du tableau. Ces cases ne contenant pas `null` ne seront pas forcément contiguës. Par exemple, le sac $\{1; 2; 1; 1; 0\}$ peut être, par exemple, représenté par le tableau de 10 cases $[null; 1; 2; null; null; 1; null; 1; 0; null]$, où les entiers sont des objets de type `Integer`.

   La méthode `ajouter(...)` mettra l'élément dans la première case nulle du tableau. Si le tableau est plein, on ne fait rien.

   La méthode `retirerUn(...)` mettra à `null` la première case correspondant à un exemplaire de l'élément donné en argument.

   On défnira un constructeur ayant un paramètre entier indiquant le nombre maximal d'éléments contenu dans le sac (et donc du tableau utilisé pour sa représentation). (2,5 pts)

4. On voudrait maintenant faire une implémentation **SacTab** de `Sac` avec un tableau mais paramétrée par un type `E`.
   Est-il possible de le représenter par un tableau de type `E` ? Si ce n'est pas le cas, comment peut-on contourner le problème et avec quels inconvénients ? Expliquez sans écrire la totalité du code.(1pt)
5. Dans la classe **SacTabInt**, on s'intéresse au traitement de `ajouter` quand le sac est plein, pour cela, définissez une exception `SacPlein`.
   Modifiez les classes précédentes de façon à ce que en cas d'ajout impossible l'exception `SacPlein` soit lancée. En maintenant la hiérarchie des

construction de la forme `for(Object o : p) ...` . (On rappelle qu'il faut pour cela que la classe implémente l'interface `Iterable` qui est décrite succinctement dans l'annexe.) La méthode `remove()` ne sera pas "implémentée".(3pts)

(c) On ajoutera également une méthode `affiche(...)` qui utilisera le point précédent.(1pts)

7. Définissez une classe **SacTabBis** extension de `SacTabUn` telle qu'en cas d'ajout dans un sac plein au lieu de lancer `SacPlein`, la taille du tableau de données pour représenter le sac soit augmentée. Pour cela, on redéfinira (entre autres choses) `ajouter` de façon à appeler `ajouter` de `SacTabUn` et à attraper l'exception `SacPlein` et, dans ce cas, augmenter la taille du tableau. (2pts)
8. On va dans ce qui suit utiliser les piles définies précédemment de façon concurrente.
   (a) Définissez une extension **SacThread** de `Thread` contenant un sac `s` et telle qu'un objet de cette classe ajoute successivement tous les entiers de 0 à 99. Cette classe

## Annexe

**Classe LinkedList <E>** implémente `Iterable<E>`

- `LinkedList()` constructeur : construit une liste vide.
- `public boolean add(E e)` ajoute l'élément o en fin de liste.
- `public E get(int index)` retourne l'élément situé à l'indice `index`.
- `public boolean remove(Object o)` enlève le premier objet égal à o s'il y en a un (la méthode `equals` est utilisée). Retourne `true` si un tel élément existe, `false` sinon.
- `public E remove(int index)` enlève l'élément situé à l'indice `index`.
- `Iterator<E> iterator()` retourne un itérateur donnant tous les objets de cette liste.

**Interface Iterator<E>** L'interface `Iterator<E>` contient ces 3 méthodes :

- `boolean hasNext()` retourne `true` si l'itérateur a encore des éléments à parcourir en allant vers l'avant.
- `E next()` retourne l'élément prochain (vers l'avant) dans la liste.
- `void remove()` élimine l'élément que l'on vient de lire avec `next()`. Opération optionnelle : lance `UnsupportedOperationException` si pas "implémentée".

**Interface Iterable<T>** L'interface `Iterable<T>` contient cette unique méthode :

- `Iterator<T> iterator()` retourne un [illegible] élément [illegible]