20/03/2021 - 13h30-15h30

Initiation à la Programmation (IP2)

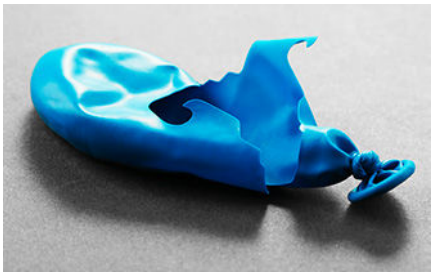
Aucun document autorisé. Dans vos réponses n'utilisez **aucune des librairies de java** qui sont en rapport avec les listes, vous devez écrire tout ce qui vous est utile.
Pensez à dire sans ambiguïté à quelle classe vos méthodes appartiennent. Ne comptez pas sur le correcteur pour le deviner : c'est à vous d'être clair.

Ce sujet est composé de 3 exercices. Ils sont indépendants, et apparaissent dans l'ordre que nous avons suivi pour la progression du cours. Le barème est indicatif, et vous donne une idée du temps à consacrer aux exercices. Si le barème évolue légèrement ce sera pour donner de l'importance aux exercices traités complètement et correctement.

Exercice 1 (5 points) On s'intéresse à la modélisation d'éléments de la vie courante sous la forme d'une classe Java : ici de simples ballons de baudruche colorés.

Voici en image la forme qu'ils peuvent avoir, lorsqu'ils vous sont remis initialement, quand ils sont finalement gonflés, ou malheureusement éclatés :





Voici ce que vous pouvez faire pour les utiliser :

- souffler dedans pour le gonfler (une fois par une fois)
- laisser un peu d'air s'échapper
- faire un noeud à son extrémité
- l'éclater

Remarquez qu'un ballon peut aussi éclater tout seul si vous le gonflez trop, et que cette pression maximum est connue à priori (disons que c'est la même pour tous)

Définissez une classe pour cet objet, écrivez la totalement en étant soigneux. Prévoyez un affichage de ces objets. Ecrivez un main supposé être dans une classe externe qui montre que vous avez été attentif à toutes les situations.

Exercice 2 (5 points)

On s'intéresse au triangle de pascal :

n \ k	0	1	2	3	4	5	6	7	8	9	10
0	1										
1	1	1									
2	1	2	1								
3	1	3	3	1							
4	1	4	6	4	1						
5	1	5	10	10	5	1					
6	1	6	15	20	15	6	1				
7	1	7	21	35	35	21	7	1			
8	1	8	28	56	70	56	28	8	1		
9	1	9	36	84	126	126	84	36	9	1	
10	1	10	45	120	210	252	210	120	45	10	1

On rappelle qu'il est construit ligne par ligne, et qu'une valeur à un endroit précis est le résultat de l'addition de deux valeurs de la ligne précédente : celle juste au dessus d'elle, et celle qui est au dessus et à sa gauche. (Sur la figure : le 35 de la ligne 7 colonne 4 est la somme du 15 au dessus et du 20 immédiatement à gauche)

Nous allons utiliser les listes pour représenter une seule ligne de ce triangle. Du coup cette règle mnémotechnique ne sera pas tout à fait la même, il vous faudra vous repérer un peu différemment. (Mais vous avez compris le résultat attendu du calcul)

1. Ecrivez une classe LignePascal (et une classe intermédiaire pour les cellules) qui modélisera une liste d'entier simplement chaînée. Faites en sorte qu'un objet de cette classe sache à quel ligne du triangle il correspond. (Ici vos classes sont réduites à la déclaration correcte des attributs et des constructeurs dont vous avez besoin)

2. Ecrivez une méthode String toString() **récursive** qui permette d'obtenir la description d'une ligne.

3. On veut transformer un objet LignePascal en un objet correspondant à la ligne suivante. Pour cela on procédera ainsi : on ajoute une nouvelle valeur 1 en tête de liste, puis, **récursivement sur les cellules** on remplacera chaque valeur par la bonne, **puis** on remplacera les autres valeurs par la suite. Ecrivez cette méthode void buildNextLine() pour LignePascal ainsi que la méthode associée dans les cellules.

4. Vous remarquerez que les valeurs sur une même ligne sont d'abord croissantes, puis décroissantes, et que parfois la valeur maximale est doublée. Ecrivez dans la classe LignePascal une méthode **itérative** void cutPointe() qui retire cette (ou ces) valeur maximale. Vous pouvez l'écrire directement ou utiliser des méthodes intermédiaires.[1] Justifiez clairement en commentaire ce que vous faite dans un français non technique pour montrer que votre code fonctionne par design et non par chance. Notez qu'un utilisateur peut vouloir appliquer cette méthode plusieurs fois de suite.

5. **(bonus)** Ecrivez la même méthode dans une version récursive.

[1] Le résultat ne sera pas vraiment une "ligne du triangle de pascal", mais cela n'a pas d'importance on veut juste que vous fassiez cette suppression correctement

Exercice 3 (10 points) Dans presque toutes vos applications (bureautique, navigateur etc ...) vous avez dans le menu de vos fenêtres deux petites flèches orientées dans le sens des aiguilles d'une montre ou dans le sens inverse. Ils vous permettent de revenir sur la dernière action effectuée. Cette fonctionnalité s'appelle selon le cas : Annuler, Défaire, Undo, Reculer, Précédent, etc ... il s'agit toujours du même genre de comportement : retourner à la situation où vous étiez juste avant de faire une action. On peut aussi naviguer dans l'autre sens avec un bouton permettant de Rétablir, Refaire, Redo, etc ....

Pour gérer cette fonctionnalité on va définir une classe StateManager qui va beaucoup ressembler aux listes qu'on a rencontré. Son rôle est de stocker dans des cellules toute la suite des actions (celles qu'on considère être réalisées et celles en attente d'être refaites par d'éventuels Redo). Le contenu de ces cellules sera une simple chaîne de caractères qui vaudra soit "actionA" soit "actionB". (On pourrait généraliser mais il est suffisant ici de ce limiter à 2 actions)

Voici sa trame, la cellule head nous permet de savoir où nous en sommes lorsque plusieurs undo/redo ont été déclenchés : elle repère la limite des actions qui **sont considérées réalisées** [2]

```
public class StateManager {
  CellState first;
  CellState head;

  public void addnewActionA() { addnewAction("actionA"); }
  public void addnewActionB() { addnewAction("actionB"); }

  private void addnewAction(String name) { ... }

  public boolean isUndoable() {...}
  public boolean isRedoable() {...}

  public String getActionBack() {
    // on suppose que le prerequis isUndoable est verifie'
    ...
  }

  public String getActionForward() {
    // on suppose que le prerequis isRedoable est verifie'
    ...
  }

  // affiche les actions A,B actuellement effectuees
  public void printToHead() {...}
}
```

[2] la cellule head contient la dernière action qui a été réalisée

Pour bien comprendre comment ça marche :

(a) Supposons qu'on vienne de faire la suite d'actions AABA. Alors la liste StateManager correspondante contiendra la suite "actionA", "actionA", "actionB", "actionA", avec head qui pointe vers la dernière action A.

(b) Si à présent on revient 2 fois en arrière, et qu'on appuie ensuite 2 fois sur le bouton redo, on revient exactement à la même situation qu'en (a)

(c) Si l'on revient simplement 2 fois en arrière, et rien d'autre, la liste StateManager contient encore ces 4 actions, mais head pointe vers le second "ActionA", ce qui permet de savoir où on en est exactement dans cette ligne temporelle.

(d) Si à présent on choisit d'effectuer une action A [3], la liste contiendra 3 fois "ActionA", avec head qui repère la référence de la 3ème cellule.

On vous donne la classe CellState :

```
public class CellState {
   private CellState prev;
   private CellState next;
   private final String actionName;
   public CellState(String n) {actionName=n; prev=null; next=null;}

   public void setNext(CellState x) { next=x;}
   public void setPrev(CellState x) { prev=x; }
   public CellState getprev() { return prev;}
   public CellState getnext() { return next; }

   public String getactionName() { return actionName;}

   public void printTo(CellState stop) { ... }
}

```

1. **(2 points)** La méthode printToHead de la classe StateManager doit afficher le contenu des cellules situées entre first et head. C'est à dire qu'elle affiche la suite des actions qu'on considère être réalisées. On veut pour cela essentiellement déléguer le travail à la classe CellState, avec sa méthode printTo et que ce travail y soit réalisé de façon **itérative**. Ecrivez ces deux méthodes.

2. **(2 points)** Les méthodes isUndoable et isRedoable de StateManager doivent permettre de savoir si actuellement il est possible d'appuyer sur undo (respectivement redo). Ecrivez ces 2 méthodes.

3. **(1 point)** La méthode getActionBack a pour rôle de retourner le nom de la dernière action faite et de considérer qu'elle a été annulée. Ecrivez cette méthode (vous pouvez supposer que isUndoAble a été testé auparavant et qu'il n'y aura pas de problèmes)

4. **(1 point)** La méthode getActionForward a pour rôle de retourner le nom de l'action à refaire et de considérer qu'elle a été faite. Ecrivez cette méthode (vous pouvez supposer que isRedoAble a été testé auparavant et qu'il n'y aura pas de problèmes)

5. **(2 points)** La méthode addnewAction est indépendante des boutons undo/redo, et ajoute une nouvelle action, conformément à la description (d). Ecrivez cette méthode.

6. **(2 points)** Nous allons mettre notre StateManager en pratique, en l'utilisant pour une classe MyObject dont voici la trame :

[3] Le traitement d'une action B serait similaire

```
public class MyObject {
  private StateManager sm;
  public MyObject(){ sm=new StateManager(); }

  public void faitA() {...}
  public void faitB() {...}
  private void realiseA() {System.out.println("A est realise");}
  private void defaitA() {System.out.println("A est defait");}
  private void realiseB() {System.out.println("B est realise");}
  private void defaitB() {System.out.println("B est defait");}

  public void undo(){...}
  public void redo(){...}
}
```

Notez que les méthodes publiques sont faitA, faitB, undo, redo, et qu'il y a 4 méthodes privées realiseA, realiseB, defaitA, defaitB qui permettent la réversibilité des actions A et B. Vous devriez comprendre aisément pourquoi.

Ecrivez les méthodes undo, redo et faitA