Aucun document. Aucune machine. Barème indicatif. Parties indépendantes.
Une question peut toujours être traitée en utilisant les précédentes (traitées ou non).
Les morceaux de code Java devront être clairement présentés, indentés et commentés.

1 Généalogie inversée

On considère une généalogie dans laquelle on regarde tous les ascendants d'une personne donnée : ses parents, leurs propres parents, etc. Une telle généalogie peut clairement être représentée par un arbre binaire donné par la classe `Personne` [1] :

```
class Personne{
    private String prenom, nom;
    private Personne mere, pere;
}
```

Dans toute cette partie (exercices 1 à 5), les méthodes demandées sont à écrire dans la classe `Personne`. Un *parent* d'une personne est son père ou sa mère. Un *ascendant* est soit un parent, soit l'ascendant d'un parent (exemples : grand-père, arrière-grand-mère, etc.).

Exercice 1. (1,5 points)

1. Écrire une méthode `dansMemeFratrie(Personne p)` qui teste si l'objet courant est un frère ou une sœur (=les deux mêmes parents) de l'instance de `Personne` passée en argument.
2. Écrire une méthode `estCousinGermain(Personne p)` qui teste si l'objet courant est un(e) cousin(e) germain(e) de l'instance de `Personne` passée en argument (c'est-à-dire si un parent de l'un est frère ou sœur d'un parent de l'autre).

Exercice 2. (2 points) Écrire une méthode `possedeCommeAscendant(Personne p)` qui teste si l'objet courant possède comme ascendant l'instance de `Personne` passée en argument.

Exercice 3. (1,5 points) On introduit une *distance* entre deux personnes : si aucune des deux n'est l'ascendante de l'autre, c'est l'infini, sinon c'est la longueur de la branche à parcourir pour passer de l'une à l'autre (soit en montant, soit en descendant dans l'arbre). Par exemple, la distance entre une personne et sa mère est 1, entre elle et son grand-père, c'est 2, avec elle-même, c'est 0, et la distance entre une personne et son oncle est infinie.

Écrire une méthode `distance(Personne p)` qui renvoie la distance entre l'objet courant et l'instance de `Personne` passée en argument.

Exercice 4. (2 points) Écrire une méthode `cEstQui(Personne p)` qui affiche ou renvoie (au choix) une chaîne de caractères décrivant le lien d'ascendance/descendance entre l'objet courant et `p`.

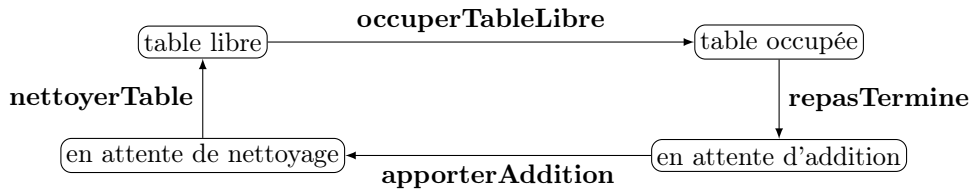
Quelques exemples : `"toi"` / `"tu es sa grand-mere"` / `"ton arriere-arriere-grand-pere"` / `"vous n'etes pas lies"`

Exercice 5. (3 points) Écrire une méthode `ascendantsDeMemePrenomADistMin()` qui renvoie la liste des ascendants de l'objet courant ayant le même prénom que l'objet courant et à distance minimum. On pourra utiliser `java.util.LinkedList`.

2 Gestion d'une salle de restaurant

L'utilisation de collections de l'API Java est interdite dans cette partie (entre autres `LinkedList` et `ArrayList`).

On veut gérer informatiquement l'occupation des tables dans une salle de restaurant (pour simplifier on ne se soucie pas du nombre de couverts par table). Une table donnée est dans un des 4 états suivants : libre, occupée, en attente d'addition, en attente de nettoyage. Le passage d'un état à un autre se fait suivant le cycle :



L'ensemble des tables de la salle est donc partagé en 4 sous-ensembles, chacun représenté par une liste. Suivant le sous-ensemble, la gestion des entrées / sorties est différente :
– tables occupées : entrée en début de liste, sortie à la demande ;
– tables en attente d'addition : file (premier entré / premier sorti) ;
– tables en attente de nettoyage : file ;
– tables libres : aucune importance.
Chaque table est identifiée par un entier (deux tables distinctes ont des identifiants distincts), et la classe `Table` est fournie ; elle possède entre autres une méthode `public int getId()` renvoyant l'identifiant de la table. **On ne demande pas d'écrire la classe `Table`.**

1. Pour simplifier la programmation, on prend une représentation "traditionnelle" de la famille : chaque personne possède un père et une mère, éventuellement inconnu(e)(s).

2.1 Listes

On veut créer deux classes : `ListeTables` qui représente le début d'une liste et `CTable` qui représente une cellule référençant une table. L'interface de `ListeTables` est :

```
/** ajoute la cellule en debut de liste
 * @param cTable cellule a ajouter */
public void addFirst(CTable cTable);
/** supprime la derniere cellule de la liste et renvoie cette cellule
 * @return cellule supprimee */
public CTable removeLast();
/** supprime la cellule referencant la table d'identifiant donne
 * et renvoie cette cellule si elle existe
 * @param idTable identifiant de la table
 * @return cellule contenant la table d'identifiant idTable */
public CTable removeCell(int idTable);
/** teste si la liste est vide */
public boolean isEmpty();
```

Exercice 6. (2 points) Donner les attributs des classes `CTable` et `ListeTables` et faites un schéma représentant une liste de quatre cellules. On rappelle qu'une liste instance de `ListeTables` doit pouvoir facilement être gérée comme une file : l'ajout d'un élément se fait en début de liste, la suppression en fin, et ces deux opérations doivent se faire en temps indépendant de la longueur de la liste.

Exercice 7. (1,5 points) Écrire les méthodes `addFirst` et `removeLast` de la classe `ListeTables`, de telle sorte que le temps d'exécution ne dépende pas de la longueur de la liste.

Exercice 8. (1,5 points) Écrire la méthode `removeCell` de la classe `ListeTables`.

2.2 Salle de restaurant

La classe `SalleRestaurant` possède 4 attributs privés de type `ListeTables` : `tablesOccupees`, `enAttenteAddition`, `enAttenteNettoyage` et `tablesLibres`. Par ailleurs elle possède l'interface suivante :

```
/** fait passer une table libre en table occupee
 * (transfert une cellule de la liste des tables libres a la liste des tables occupees)
 * @return true s'il y avait une table libre, false sinon */
public boolean occuperTableLibre();
/** fait passer une table occupee en table en attente d'addition
 * @param idTable identifiant de la table qui a termine son repas
 * @return true si cette table etait occupee, false sinon */
public boolean repasTermine(int idTable);
/** fait passer une table en attente d'addition en table en attente de nettoyage
 * @return true s'il y avait une table en attente d'addition, false sinon */
public boolean apporterAddition();
/** fait passer une table en attente de nettoyage en table libre
 * @return true s'il y avait une table a nettoyer, false sinon */
public boolean nettoyerTable();
```

Exercice 9. (1 point) On suppose que la salle possède 12 tables : 3 sont libres (tables 3, 7 et 11), 5 occupées (tables 1, 2, 8, 9 et 12), 2 en attente d'addition (tables 4 et 6) et 2 en attente de nettoyage (tables 5 et 10). Faire un schéma de la structure de données.

Exercice 10. (4 points) Écrire les méthodes de la classe `SalleRestaurant` (explicitées ci-dessus).

Exercice 11. (2 points) Pour attribuer une table à un nouveau client, on suit l'algorithme suivant :
- s'il existe une table libre, on la lui attribue,
- sinon, s'il existe une table en attente de nettoyage, on la nettoie, puis on l'attribue au client,
- sinon, s'il existe une table en attente d'addition, on apporte l'addition, on nettoie la table, puis on l'attribue au client,
- sinon on lui dit que ce n'est pas possible.

Écrire la méthode suivante de la classe `SalleRestaurant` :

```
/** attribue une table a un nouveau client
 * @return la cellule associee a la table attribuee si c'est possible, null sinon */
public CTable attribuerTable();
```