IPO – TP 2 : premières classes

http://www.lri.fr/~blsk/IPO/

Tableaux décalés Dans certains langages de programmation, on définit pour chaque tableau son indice de début et son indice de fin. Ainsi, on peut déclarer un tableau de longueur 4 comme allant de l'indice 2 (inclus) à l'indice 6 (exclu), ou même de l'indice -2 (inclus) à l'indice 2 (exclu).

On simule ceci à l'aide d'une classe `Tab` dont les attributs sont un tableau primitif Java de la taille voulue, et les indices de début (inclus) et de fin (exclu). On considère que notre tableau contient des chaînes de caractères. Définir une telle classe `Tab` pour qu'elle contienne :

1. les bonnes déclarations d'attributs,
2. un constructeur prenant en paramètres un indice de début (inclus) et un indice de fin (exclu),
3. une méthode `length()` donnant la longueur du tableau,
4. une méthode `get(int i)` renvoyant l'élément du tableau à l'indice `i` (en tenant compte du décalage : si le premier indice du tableau est 3, un accès `get(3)` renverra le premier élément, un accès `get(4)` le deuxième élément, etc.),
5. une méthode `set(int i, String s)` plaçant la chaîne `s` à l'indice `i` (en tenant compte du décalage),
6. une méthode `main` exécutant ces opérations sur quelques exemples.

Si, lors de la construction, l'utilisateur fournit les bornes dans le mauvais ordre, on créera un tableau de longueur zéro. Si les méthodes `get` et `set` reçoivent en paramètre un indice invalide, on laissera se déclencher l'erreur `ArrayIndexOutOfBoundsException` de Java.

Fractions Définir une classe `Fraction` dont chaque objet est une fraction représentée par son numérateur et son dénominateur. On demande que le dénominateur soit strictement positif. La classe devra comporter les méthodes suivantes :

1. une méthode `String toString()` qui renvoie une représentation de la fraction sous la forme `n / d`, où `n` est le numérateur et `d` le dénominateur ; dans le cas où le numérateur vaut `1`, on affichera simplement `n`,
2. deux méthodes `Fraction add(Fraction f)` et `Fraction mul(Fraction f)` qui renvoient chacune une *nouvelle* fraction, égale à la somme ou au produit de **`this`** et `f`,
3. une méthode **`boolean`** `egale(Fraction f)` qui renvoie **`true`** si la fraction `f` est (numériquement) égale à la fraction courante,
4. une méthode **`int`** `compareTo(Fraction f)` qui renvoie un entier qui est :
   - positif si **`this`** est strictement plus grande que `f`,
   - négatif si **`this`** est strictement plus petite que `f`,
   - zéro si **`this`** et `f` sont égales.
5. une méthode `main` qui teste toutes ces opérations sur quelques exemples.

Bonus : faire en sorte que les fractions soient toujours réduites, c'est-à-dire que le numérateur et le dénominateur n'aient pas de facteur non trivial commun.

Chronomètre On veut créer une classe `Chrono` représentant un temps, mesuré en heures, minutes et secondes. Définir une telle classe avec les éléments suivantes :

1. trois attributs entiers pour les heures, minutes et secondes, et un constructeur,
2. une méthode `String toString()` qui donne d'un temps un affichage agréable,
3. une méthode `int toSeconds()` qui convertit un temps en un nombre de secondes,
4. une méthode `void normalise()` qui modifie un objet de la classe `Chrono` de sorte à s'assurer que les secondes et les minutes appartiennent toujours à l'intervalle $[0, 59]$,
5. une méthode `boolean equals(Object o)` qui renvoie `true` si l'objet passé en paramètre représente le même temps que `this`,
6. une méthode `boolean avant(Chrono c)` qui renvoie `true` si le temps représenté par `this` est inférieur ou égal au temps `c`,
7. une méthode `void avance(int n)` qui fait avancer un temps de `n` secondes,
8. une méthode `Chrono diff(Chrono c)` qui renvoie un *nouvel* objet `Chrono` représentant l'écart de temps entre `c` et `this`,
9. une méthode `main` testant toutes ces opérations sur quelques exemples.

Intervalles On veut définir une classe `Intervalle` représentant des intervalles de temps. Un intervalle est défini par deux objets de la classe `Chrono`, représentant respectivement le début et la fin de l'intervalle. Définir une telle classe avec les éléments suivants :

1. des attributs représentant le temps de début et le temps de fin,
2. un constructeur prenant en paramètres deux temps, qui fera en sorte de créer un intervalle de durée zéro si les bornes ne sont pas fournies dans le bon ordre,
3. une méthode `String toString()` renvoyant une chaîne de caractères représentant l'intervalle,
4. une méthode `Chrono duree()` renvoyant un temps représentant la durée de l'intervalle,
5. une méthode `boolean avant(Intervalle i)` qui renvoie `true` si l'intervalle `this` est intégralement avant l'intervalle `i`.
6. une méthode `boolean conflit(Intervalle i)` qui renvoie `true` si `this` et `i` ont une intersection non vide,
7. une méthode `main` testant toutes ces opérations sur quelques exemples.

Agenda On appellera un *agenda* une collection d'intervalles deux à deux disjoints. On veut créer une classe `Agenda` avec un attribut de type `ArrayList<Intervalle>`, comportant des intervalles disjoints rangés par ordre chronologique. Définir une telle classe avec les éléments suivants :

1. le tableau attribut, et un constructeur créant un agenda vide,
2. une méthode `String toString()` affichant l'intégralité d'un agenda,
3. une méthode `boolean compatible(Intervalle i)` qui renvoie `true` si l'intervalle donné en paramètre est bien disjoint de tous les intervalles déjà présents dans l'agenda,
4. une méthode `boolean insertion(Intervalle i)` qui insère l'intervalle `i` dans l'agenda `this` si cela est possible (et dans ce cas renvoie `true`), ou qui renvoie `false` sinon.
5. une méthode `main` testant toutes ces opérations sur quelques exemples.