01234567890123456789012345

Université d'Aix-Marseille III - L2 Math-Info
I4 - Durée : 1h30 - ni document, ni calculatrice autorisés

Important : dans toutes les classes que vous écrirez, tout attribut devra être déclaré privé.

1. Un particulier loue son appartement toute l'année à des touristes. Les séjours se réservent pour un nombre quelconque de jours, en indiquant le jour d'arrivée et le jour de départ.

1.1 *Définir la classe* `Sejour` *contenant les méthodes suivantes :*

- un constructeur prenant deux entiers en paramètre : les numéros des jours de départ et d'arrivée (1er janvier : 1, 31 décembre : 365).
- la méthode `boolean chevauche(Sejour s)` qui indique si `s` chevauche ce séjour (il y a chevauchement si le début ou la fin du premier séjour se trouve entre le début et la fin du second).
- la méthode `Sejour compromis(Sejour s)` qui retourne un compromis de séjour quand `s` chevauche ce séjour : la fin de `s` est avancée au jour précédent le début de ce séjour, sinon le début de `s` est repoussé le jour suivant la fin de ce séjour, sinon il n'y a pas de compromis possible.

1.2 *Définir la classe* `Reservations` *qui contient la méthode* `Sejour reserve(Sejour s)`. Cette méthode mémorise et retourne un nouveau (compromis de) séjour, si ç'a été possible. Elle retourne `null` sinon. Remarque : `s` peut chevaucher au maximum deux séjours précédemment mémorisés. Le compromis de séjour est alors un compromis fait avec chacun de ces deux séjours.

2. On veut définir des électeurs, qui peuvent choisir le candidat pour qui voter, et des candidats, qui sont aussi des électeurs. Nous allons donc définir une classe `Electeur` et une classe `Candidat`, qui héritera de `Electeur`.

2.1 *Définir la classe* `Electeur`*, qui mémorise le nom (de type* `String`*) de l'électeur et son candidat (de type* `Candidat`*, défini dans la question* **2.2***), et définit les méthodes suivantes : le constructeur* `Electeur(String nom)`*, l'accesseur* `String getNom()`*, la méthode* `void choisit(Candidat c)` *qui mémorise le candidat choisi, et la méthode* `String sonCandidat()` *qui retourne le nom du candidat choisi.*

2.2 *Définir la classe* `Candidat`*, qui hérite de* `Electeur`*, définit le constructeur* `Candidat(String nom)` *et définit la méthode* `void rencontre(Electeur e)`. Cette méthode fait en sorte que l'électeur `e` choisit ce candidat s'il n'en a pas encore choisi. Par ailleurs, on fera en sorte que le candidat se soit automatiquement choisi lui-même comme candidat.
Exemple de mise en oeuvre des classes `Electeur` et `Candidat` :

```
Electeur e = new Electeur("Martin");
Candidat c = new Candidat("Sarkoyal");
c.rencontre(e);
System.out.println(e.sonCandidat() + " " + c.sonCandidat()); // affiche "Sarkoyal Sarkoyal"
```

3. Nous considérons un monde manichéen où l'ensemble des individus est divisé entre les militants "quichistes", ceux qui aiment les quiches, et les militants "anti-quichistes", ceux qui n'aiment pas les quiches. Toutes les classes de la partie **3** implémenterons l'interface suivante :

```
interface Militant {
   String proclamation();
   boolean estQuichiste();
}
```

3.1 *Définir la classe* `Quichiste` *dont la méthode* `String declaration()` *retourne la chaîne de caractères "J'aime les quiches !"* et la méthode `boolean estQuichiste()` retourne `true`.

3.2 Si ce monde était aussi simple qu'il en avait l'air a priori, nous n'aurions plus qu'à définir une classe pour les anti-quichistes. Malheureusement, pour se distinguer, certains militants se désignent anti-anti-quichistes, anti-anti-anti-quichistes, etc, bref rajoutent un certain nombre de fois le préfixe "anti" à "quichiste". Cependant, cela ne sert à rien car dans ce monde, être anti-anti-X c'est équivalent à être X.
Définir la classe `Anti`*, qui implémente* `Militant` *et permet de définir un anti-X, à partir de X, un militant d'un certain type.* Pour ce faire, cette classe définira un constructeur prenant en paramètre un militant X. Le militant anti-X ainsi créé proclamera "Il est faux que :" suivi de ce que proclame X, et la méthode `boolean estQuichiste()` retournera la négation de ce qu'elle retourne pour X.
Exemple d'utilisation des classes `Anti` et `Quichiste` :

```
Militant aaq = new Anti(new Anti(new Quichiste()));
System.out.println(aaq.proclamation());
// affiche "Il est faux que : Il est faux que : J'aime les quiches !"
```

3.3 Pour encore plus se distinguer, certains militants insèrent plusieurs fois le préfixe "pro" dans leur désignation. Ex : pro-anti-anti-pro-quichiste. Là encore, ils compliquent les choses pour rien, car être pro-X, c'est simplement être X.
Définir la classe `Pro`*, qui permet de définir un pro-X, dans le même esprit qu'à la question* **3.2**. Dans sa proclamation, un pro-X se contente de rajouter "Il est vrai que :".

3.4 Certains militants ont conscience que leur proclamation est délicate à interpréter. C'est pourquoi ils ajoutent à la fin de leur proclamation habituelle "En clair :" suivi de ce que déclarerait un simple quichiste ou anti-quichiste.
Définir la classe `Eclaircisseur` *qui permet d'éclaircir la proclamation de n'importe quel type de militant.* Cette classe aura un constructeur qui prend en paramètre le militant dont il faut éclaircir la proclamation.
Exemple d'utilisation de la classe `Eclaircisseur` :

```
Militant epaaq = new Eclaircisseur(new Pro(aaq));
System.out.println(epaaq.proclamation());
// affiche "Il est vrai que : Il est faux que : Il est faux que : J'aime les quiches !
   En clair : J'aime les quiches !"
```

4. Nous allons maintenant prendre en compte le fait que certains électeurs sont aussi des militants. Comme nous avons déjà défini les classes `Electeur` et des classes implémentant `Militant`, nous allons pouvoir simuler l'héritage multiple, en créant des classes qui héritent de `Electeur` et qui implémentent `Militant`.

4.1 *Définir la classe* `ElecteurMilitant` *qui hérite de* `Electeur` *et implémente* `Militant`. Le constructeur de cette classe prend en paramètre un nom et un militant d'un certain type. Le comportement d'une instance de cette classe est à la fois celle d'un électeur et celle du type de militant choisi lors de sa création, à ceci près qu'elle ne peut choisir un candidat que de son camp (quichiste ou anti-quichiste).

4.2 *Définir la classe* `Politicien` *qui hérite de* `Candidat` *et implémente* `Militant`. Le constructeur de cette classe prend en paramètre un nom et un militant d'un certain type. Le comportement d'une instance de cette classe est à la fois celle d'un candidat et celle du type de militant choisi lors de sa création, sauf qu'il dispose d'une méthode supplémentaire : `float popularite()` permet de connaître le pourcentage d'électeurs qui voteraient pour lui. Ce nombre, compris entre 0 et 1, est égal au nombre d'électeurs ayant choisi ce candidat divisé par le nombre d'électeurs ayant choisi un candidat. Exemple :

```
ElecteurMilitant m = new ElecteurMilitant("Martin", aaq);
Politicien p1 = new Politicien("Sarkoyal", aaq);
Politicien p2 = new Politicien("Rozy", epaaq);
p1.rencontre(m);
System.out.println(p1.popularite()); // affiche "0.667" (Martin+Sarkoyal contre Rozy)
```