

Titre :
(LgP6Linf.EPS)
Auteur :
Adobe Illustrator(TM) 3.2
Aperçu :
Cette image EPS n'a pas été enregistrée
avec un aperçu intégré.
Commentaires :

Examen de juin écrit

Durée : **2h00** - Tous les documents *personnels*, manuscrits ou imprimés, sont autorisés.
Tous les appareils électroniques sont interdits et doivent être rangés.

NE RIEN ÉCRIRE AU CRAYON NI À L'ENCRE ROUGE S.V.P.

Ce sujet comporte cinq pages.

I Questions de cours (6 points)

I-A On considère les deux classes suivantes :

```
class Filtre {
    protected static int nbObj= 0;

    Filtre() { super(); nbObj++; }

    public Filtre selonD(double d) {
        return (d<0.5)?null:this;
    }
}

class Test {
    public static void main(String[] a) {
        // utilisation de Filtre
        for (int i= 0; i<5; i++){
            System.out.println(Filtre.nbObj+" "+(new Filtre()).selonD(Math.random()));
        }
    }
}
```

q1 Quelle est la signification java de la ligne de la classe Filtre

```
protected static int nbObj= 0;
```

Quelle est l'utilité d'une telle déclaration pour ce programme ?

q2 Décortiquez la ligne de la classe Test

```
System.out.println(Filtre.nbObj+" "+(new Filtre()).selonD(Math.random()));
```

Indiquez ses éléments constitutifs et expliquez chacun d'eux.

q3 On considère que les valeurs successives fournies à l'exécution par `Math.random()` sont

```
0.21245245192316065
0.10849780657470653
0.9932333063646892
0.12333740384282699
0.5545378339379208
```

Qu'est-ce qui s'affiche à l'écran (Les valeurs précises dépendantes de la plateforme matérielle et logicielle et de l'exécution seront indiquées par un indication du type `< valeur >`) ? Justifiez alors votre résultat et indiquez à quoi correspondent ces `< valeur >`.

I-B On considère les quatre classes suivantes :

```
abstract class M {
    protected abstract void m();
}

abstract class F extends M{
    public String toString() {
        return "Depuis F : "+super.toString();
    }
}

class PF extends F{
    public void m() {
        System.out.println("Depuis PF : methode m");
    }

    public String toString() {
        return "Depuis PF : "+super.toString();
    }
}

class Test {
    public static void main(String[] a) {
        // utilisation de M, F et PF
        M b= new PF();
        b.toString();
        System.out.println();
        b.m();
        System.out.println(b);
    }
}
```

q4 Pourquoi la classe M est-elle nécessairement déclarée comme abstraite ?

Pourquoi la classe F est-elle nécessairement déclarée comme abstraite ?

q5 Relativement aux autres classes, que peut-on dire à propos de la méthode de la classe PF

```
public void m() { ... }
```

q6 Relativement aux autres classes, que peut-on dire à propos de la méthode de la classe PF

```
public String toString() { ... }
```

Même question pour la méthode de même nom de F.

q7 Que signifie l'affectation de la classe Test

```
M b = new PF();
```

Généralisez vos explications en indiquant ce que veut dire :

```
<nom de classe> <identificateur> = new <nom de classe> ();
```

q8 Qu'est-ce qui s'affiche à l'écran ? Justifiez votre résultat.

II Problème (14 points)

Présentation du problème

L'association à but non lucratif AMVA s'est fixé pour objectif d'aider les aveugles pour les tâches de la vie quotidienne qu'ils ne peuvent assumer seuls du fait de leur handicap. En particulier, elle se charge de fournir aux aveugles des guides pour leurs déplacements dans la ville. Pour cela elle dispose d'une liste de volontaires qui acceptent de guider les aveugles d'un lieu à un autre, à condition que le déplacement de l'aveugle ne soit pas trop loin de leur domicile.

Les aveugles qui désirent bénéficier des services offerts par l'association doivent obligatoirement être membres de cette association. Pour simplifier, on supposera que les bénévoles n'en sont pas membres. Bien entendu la gestion de l'association coûte de l'argent, et pour avoir les fonds nécessaires il existe aussi des membres bienfaiteurs, dont la cotisation est supérieure à celle des aveugles et qui ne bénéficient pas des services de l'association.

Lorsqu'un aveugle désire se déplacer d'un lieu à un autre, il téléphone à la permanence de l'association et il indique à quel endroit il désire se rendre à partir de son domicile (restriction par rapport à la réalité). On établit alors une fiche de **mission**, comportant une référence de l'aveugle, les lieux de départs et d'arrivée de la mission. On recherche alors un guide bénévole qui puisse réaliser la mission et on indique son nom sur la fiche de mission. Pour pouvoir fonctionner, l'association répertorie, entre autres, tous ses membres aveugles, les guides bénévoles et les missions mettant en relation l'aveugle demandeur et le guide.

Aveugle, guide bénévole et bienfaiteur sont tous trois des *Personne*. Les deux premières sont les acteurs du guidage (*acteurGuidage*) : l'*Aveugle* qui a besoin d'un guide *Benevole* pour se déplacer; la troisième ne concerne pas le guidage : c'est le *Bienfaiteur*.

Par ailleurs, pour décrire une mission et l'association on définira les classes *Mission* et *Association*.

On dispose aussi d'un interface de constantes *IConst*.

Très important Toute indentation mal faite ou incorrecte sera notée négativement. Il est de votre responsabilité de mettre les bons attributs de champs et de méthodes, *private*, *protected* ou bien *public*.

A dessin de la hiérarchie des classes

Dessinez la hiérarchie de toutes ces classes, y compris *Object*. L'interface *IConst* n'en fait pas partie.

L'interface de constantes est le suivant :

```
interface IConst {
    final int MAX_AVEUG= 9, MAX_BENEV= 5;
    final double COTIS_NORMALE= 40.0, COTIS_BIENF= 60.0;
    final double[][] DISTANCES= { {0.},
                                    {4.13,0.},
                                    {3.86,0.87,0.},
                                    {3.69,1.63,1.52,0.},
                                    {3.11,2.39,2.09,1.30,0.},
                                    {1.74,4.76,4.46,3.04,1.09,0.},
                                    {8.26,6.09,4.87,5.65,2.15,7.26,0.},
                                    {4.86,3.92,3.26,3.04,5.00,3.80,3.59,0.},
                                    {7.17,3.26,3.04,1.96,4.78,7.39,7.26,4.78,0.},
                                    {4.13,1.74,2.61,1.12,3.87,5.21,6.09,5.43,4.56,0.}
                                } ;
}
```

Le tableau de tableaux **DISTANCES** correspond à la matrice triangulaire des distances entre les lieux où peuvent résider et se rendre les aveugles et les bénévoles : on a donc ici dix lieux possibles. Comme dans ce texte on ne s'intéresse pas à l'algorithme de choix du « meilleur » bénévole pour une mission donnée, ce tableau de distances ne sera par la suite utilisé que pour connaître le nombre de lieux.

B classe *Association*

La classe association se présente comme suit:

```
class Association implements IConst{
    private Aveugle[] aveugles;
    private Benevole[] benevoles;
```

```

private Mission[] missions;

Association() {
    ... // creation des trois objets tableaux
    CreeAveug(); CreeBenev();
}

private void CreeAveug() {
    for (int a=0; a<aveugles.length; a++)
        aveugles[a]= new Aveugle("Av"+Integer.toString(a),
(int)(Math.random()*DISTANCES.length));
}
private void CreeBenev() {
    For (int b=0; b<benevoles.length; b++)
        benevoles[b]=new
Benevoles("Bv"+Integer.toString(b), (int)(Math.random()*DISTANCES.length));
}

public String toString() {
    ...
}
}

```

Donner la déclaration complète du constructeur par défaut de la classe Association.

Donner la déclaration complète de la méthode toString de la classe Association, dont le résultat correspond à un affichage en continu des aveugles, puis des bénévoles en continu après passage à la ligne.

C classe *Personne*

Elle comporte le nom de la personne, initialisé par le constructeur, et une méthode toString retournant le nom de la personne.

Pour que les *membres* de l'association puissent tous disposer d'un numéro d'adhérent unique, cette classe comportera un champ de classe de type entier initialisé à 0, qui sera utilisé et mis à jour dans les classes Aveugle et Bienfaiteur.

Donner la déclaration complète de la classe Personne.

D classe *acteurGuidage*

Elle comporte le lieu de **residence** (un entier !), un booléen **missionPrevue** indiquant si l'aveugle ou le bénévole est ou non déjà occupé par une mission (initialisé à false à la création de l'objet), un constructeur ayant pour paramètre le nom de la personne, et une méthode abstraite **signaler**, ayant un paramètre de type Mission.

Donner la déclaration complète de la classe *ActeurGuidage*.

E classe *Benevole*

Elle hérite de ActeurGuidage. Le constructeur a deux paramètres : le nom de la personne et son lieu de résidence (un entier !).

Elle redéfinit la méthode toString, qui utilise la méthode toString de la classe mère pour connaître le nom de la personne, en y adjoignant la qualité de la personne, "benevole".

Pour l'instant, la méthode **signaler** sera redéfinie et rendue « concrète » sans préciser les instructions dans le corps de la méthode (sa définition complète sera réalisée après la définition de la classe Mission).

Donner la déclaration complète de la classe Benevole.

F classe *Aveugle*

Un aveugle est membre de l'association et à ce titre il doit payer une cotisation et possède un numéro d'adhérent. Le champ **cotisation** comporte le montant de la cotisation. La valeur du champ **numero** est attribuée automatiquement à partir du champ de classe de la classe Personne.

Les paramètres du constructeur seront évidemment les mêmes que ceux du constructeur de *Benevole*, et le montant de la cotisation sera directement issu de l'interface *ICost* (*COTIS_NORMALE*). Le constructeur attribue un numéro.

Redéfinir la méthode *toString*, qui utilise la méthode *toString* de la classe mère pour connaître le nom de la personne, en y adjoignant la qualité de la personne, " aveugle".

Pour l'instant, vous redéfinirez concrètement la méthode *signaler* de la même façon que vous l'avez fait pour la classe *Benevole*.

Donner la déclaration complète de la classe *Aveugle*.

G classe *Bienfaiteur*

Un bienfaiteur est membre de l'association et à ce titre il doit payer une cotisation et possède un numéro d'adhérent. Le champ *cotis* comporte le montant de la cotisation et le champ *numer* le numéro d'adhérent, attribué automatiquement.

Le paramètre du constructeur correspond au nom du bienfaiteur. Le montant de la cotisation sera directement issu de l'interface *ICost* (*COTIS_BIENF*). Le constructeur attribue un numéro.

Redéfinir la méthode *toString*, qui utilise la méthode *toString* de la classe mère pour connaître le nom de la personne, en y adjoignant la qualité de la personne, " bienfaiteur".

Donner la déclaration complète de la classe *Bienfaiteur*.

H classe *Mission*

Suite à la demande initiale d'un aveugle, une mission met en relation un aveugle et un bénévole, d'un point de départ à un point d'arrivée : il y aura donc quatre champs de données pour cette classe.

Le constructeur de *Mission* n'a qu'un seul paramètre, de type *Aveugle*. Lorsqu'il s'exécute, l'objet de *Mission* est déjà créé, mais ses champs ne sont pas initialisés : c'est la méthode d'objet *signaler* de la classe *Aveugle* qui se charge d'initialiser les champs d'objet de *Mission*, à partir de ses propres données et à l'aide des modificateurs de *Mission* (cf. juste ci-dessous).

Pour pouvoir être renseignée, la classe *Mission* a besoin de trois « modificateurs » : *setDepart*, *setArrivee* et *setBenevole*.

Donner la déclaration complète de la classe *Mission*.

La méthode *signaler* de la classe *Aveugle* effectue les opérations suivantes :

Indiquer comme départ de la mission la résidence de l'aveugle.

Déterminer aléatoirement un lieu différent de la résidence et

l'indiquer comme point d'arrivée de la mission

Donner la déclaration complète de la méthode *signaler* de la classe *Aveugle*.

I compléter la classe *Benevole*

Donner la déclaration complète de la méthode *signaler* de la classe *Benevole*.