2i002 – Examen final noté sur 60 pts

Durée : 2 heures

Aucun document autorisé
– Barême indicatif –

Questions de cours (29 pts)

Exercice 1 – Lapin(s) (11 pts)

```
1  public class Lapin {
2      private String nom;
3      public int age;
4      public Lapin(String nom, int age) {
5        this.nom = nom;
6        this.age = age;
7      }
8      public boolean equals(Lapin lapin){
9        return age == lapin.age;
10     }
11     public void vieillir() {
12       age ++;
13     }
14     public static void main(String[] args){
15         Lapin  l1 = new Lapin("toto", 1);
16         Lapin  l2 = new Lapin("titi", 1);
17         Object l3 = new Lapin("tata", 1);
18         Object l4 = l1;
19
20         System.out.println(l1.equals(l2)+"␣"+l2.equals(l1));
21         System.out.println(l1.equals(l3)+"␣"+l3.equals(l1));
22         System.out.println(l1.equals(l4)+"␣"+l4.equals(l1));
23         System.out.println(l2.equals(l4)+"␣"+l4.equals(l2));
24     }
25 }
```

Q 1.1 (1 point) Donner le code du constructeur à un seul argument (`nom`) qui initialise l'âge à 0. Vous utiliserez obligatoirement l'instruction `this()`.

Q 1.2 (2 points) Donner les affichages associés à l'exécution du code du `main`. Pour chaque ligne, vous indiquerez le numéro de la ligne, et l'affichage correspondant.

Q 1.3 (2 points) Donner une version correcte de la méthode `equals`. Pour simplifier, on fait l'hypothèse que les `String` ne sont jamais `null`.

Q 1.4 (1 point) Donner la **signature** de la méthode d'instance `reproduire` qui permet à un lapin de se reproduire avec un autre lapin. Note : le principe est le même que pour l'addition entre vecteurs ou entre points.

Q 1.5 (1 point) Donner le code de la méthode `reproduire` qui effectue le travail suivant :

- générer un `Lapin` dont le nom est la concaténation des noms des parents (dans un ordre quelconque) et l'âge est 0 si **toutes** les conditions suivantes sont réunies :
  - les parents ne correspondent pas à la même instance ;
  - les parents ont le même âge et cet âge est non nul ;
  - les parents ont un nom qui commence par la même lettre (dans la classe `String`, vous utiliserez la méthode `charAt(int position)`).
- **sinon**, elle retourne la valeur `null`.

Q 1.6 (3 points) Comptage d'instances.

- Donner le nombre d'instances de `Lapin` en mémoire à l'issue de l'exécution de la **première colonne**.
- Puis, donner le **nombre total d'instances créées** à l'issue de l'exécution des **deux colonnes**.
- Enfin, donner le **nombre d'instances restant** en mémoire à l'issue de l'exécution des **deux colonnes**.

```
Lapin[] tab = new Lapin[10];
tab[0] = new Lapin("toto", 1);
tab[1] = new Lapin("titi", 1);
tab[2] = tab[0].reproduire(tab[1]);
tab[2].vieillir();
tab[3] = tab[2].reproduire(tab[0]);
tab[4] = tab[3];
for(int i=0; i<5; i++) tab[i].vieillir();
tab[5] = tab[3].reproduire(tab[4]);
```

```
Lapin parent1 = tab[0];
Lapin parent2 = tab[1];
tab[0] = new Lapin("lapinou", 3);
tab[1] = new Lapin("laplap", 4);
for(int i=5; i<10; i++)
    tab[i] = parent1.reproduire(parent2);
tab[6] = null;
tab[8] = null;
```

Exercice 2 – Exceptions (7 pts)

On considère la classe `TestLectureFichier`.

```
1 import java.io.*;
2 public class TestLectureFichier {
3   public static void main(String [] args) throws IOException {
4   int v =0;
5   File f = new File("data.txt");
6   try {
7         FileInputStream in = new FileInputStream(f);
8         int c = in.read();
9         while (c != -1) {
10                if (c == '!')
11                      v +=1;
12                if (v > 2)
13                      throw new IOException("Trop de !");
14                c = in.read();
15        }
16  }
17  catch (Exception e) { System.err.println("\nErreur " + e); }
18  finally { in.close(); }
19  System.out.println("v= "+v);
20  }
21 }
```

Le fichier `data.txt` (qui se trouve dans le bon répertoire pour pouvoir être lu) contient les 3 lignes suivantes :

```
Hello there!
...
General Kenobi!!!
```

Q 2.1 (2 points) Lors de la compilation de cette classe `TestLectureFichier`, une erreur est signalée. Donner cette erreur et expliquer comment corriger le code de cette classe afin qu'elle puisse se compiler correctement.

Q 2.2 (2 points) Une fois corrigée, la classe `TestLectureFichier` a été compilée. Quel est le résultat obtenu par son exécution ?

Q 2.3 (2 points) On efface le fichier `data.txt` et on relance l'exécution de la classe précédente. Une exception inattendue est alors levée. Laquelle ? Proposer une correction de la classe afin qu'elle ait un comportement adéquate.

Q 2.4 (1 point) Pourquoi est-il nécessaire de spécifier `throws IOException` dans la signature de la fonction `main` ?

Exercice 3 – Programme mystère (11 pts)

```
1 public class Obj {
2     private int i,j;
3     public Obj(int i, int j) {
4         this.i = i;
5         this.j = j;
6     }
7     public void maj(Obj o){
8         System.out.println("maj");
9         i += o.i;
10    }
11    public int getResult(){
12        System.out.println("res");
13        return i+j;
14    }
15 }
```

```
16 public class ObjFils extends Obj{
17     private int k;
18     public ObjFils(int i, int j) {
19         super(i, j);
20         this.k = i+j+(int) Math.random();
21     }
22     public ObjFils(int m) {
23         this(m, m+1);
24     }
25     public int getResult(){
26         System.out.println("res (fils)");
27         return k+4;
28     }
29 }
```

```
30 public class Mystere {
31     public static void main(String[] args) {
32         Obj[] tab = {new Obj(1,2), new ObjFils(0),new ObjFils(2,1)};
33         for(int i=0; i<tab.length; i++) tab[i].maj(tab[tab.length-1-i]);
34         int s = 0;
35         for(Obj o:tab) s+=o.getResult();
36
37         System.out.println(s);
38     }
39 }
```

Q 3.1 (1 pt) Dessiner un diagramme mémoire correspondant à la fin de l'exécution du code sans préciser les valeurs numériques des attributs.

Q 3.2 (1 pt) Que penser du code `(int) Math.random()` ?

Q 3.3 (3 pts) Donner les affichages produits par ce programme.

Q 3.4 (5 pts) On propose d'ajouter le code suivant dans la classe `ObjFils` :

```
1 // Classe ObjFils
2     public void maj(ObjFils of){
3         System.out.println("maj␣(fils)");
4         super.maj(of);
5         k += of.k;
6     }
```

Q 3.4.1 (1 pt) Est-il nécessaire d'inverser les lignes 3 et 4 pour permettre la compilation ?

Q 3.4.2 (2 pts) A la fin de l'exécution du code suivant, que valent les attributs `i,j,k` de l'objet `of` ?

```
1 ObjFils of = new ObjFils(1);
2 of.maj(new Obj(1,1));
3 of.maj(new ObjFils(1,1));
```

Q 3.4.3 (1 pt) Que se passe-t-il si on oublie `super.` à la ligne 4 ? (Réponse 1 : pas d'impact, réponse 2 : ça change tout -dans ce cas expliquer-)

Q 3.4.4 (1 pt) L'ajout de cette méthode modifie-t-il les affichages du programme précédent ? Dans l'affirmative, donner les nouveaux affichages (sauf pour la ligne 37, pas besoin de refaire les calculs).

Modélisation d'une station de ski (≈30 pts)

Les exercices du problème sont relativement indépendant : même si vous n'avez pas répondu à certaines questions, vous pouvez utiliser les classes précédentes en vous inspirant de l'énoncé.

Exercice 4 – La gestion du temps par singleton (3 pts)

La classe `Temps` répond à l'ensemble des spécifications suivantes :

- Le temps est un compteur entier (géré en attribut, dont l'unité est la minute), initialisé à 0 lors de la création de l'instance.
- Nous souhaitons interdire la création d'instance de `Temps` en dehors de classe `Temps`.
- Une instance est créée en interne et le client externe interagit avec cette instance via deux méthodes de classe. De l'extérieur, la classe est utilisée comme suit :

```
1 int time = Temps.getTemps();      // Donne le temps courant
2 Temps.avancer();                  // Incrémente le compteur de temps universel
```

Q 4.1 (3 pts) Donner le code de la classe `Temps`.

Exercice 5 – Gestion des forfaits et factory (4 pts)

La classe `Forfait` répond à l'ensemble des spécifications suivantes :

- Elle contient 3 constantes à usage interne : `DEMI_JOURNEE` (240 minutes), `JOURNEE` (480 minutes), `HEURE` (60 minutes).
- Un attribut entier `int tmax` donne le temps auquel expire le forfait.
- Le constructeur reçoit en argument le temps `tmax` auquel le forfait expire.
- Il est impossible d'invoquer le constructeur en dehors de la classe.
- Trois méthodes de classe sans argument permettent d'instancier des forfaits, respectivement pour une heure, une demi-journée et une journée.
  Le temps limite correspond au temps courant (récupéré à l'aide de la classe `Temps`) plus la durée du forfait.
- Une méthode d'instance `public boolean estFini()` permet de savoir si le forfait a expiré.

Q 5.1 (3.5 pts) Donner le code de la classe `Forfait`.

Q 5.2 (0.5 pts) Donner la ligne de code permettant d'instancier un `Forfait` demi-journée depuis le programme principal.

Exercice 6 – Gestion des personnes (6.5 pts)

Les **personnes** seront ici déclinées en **skieurs** ou **surfeurs** par héritage. Dans le futur, il est envisageable d'introduire d'autres types d'acteurs dans le système. Les personnes doivent toutes répondre aux spécifications suivantes :

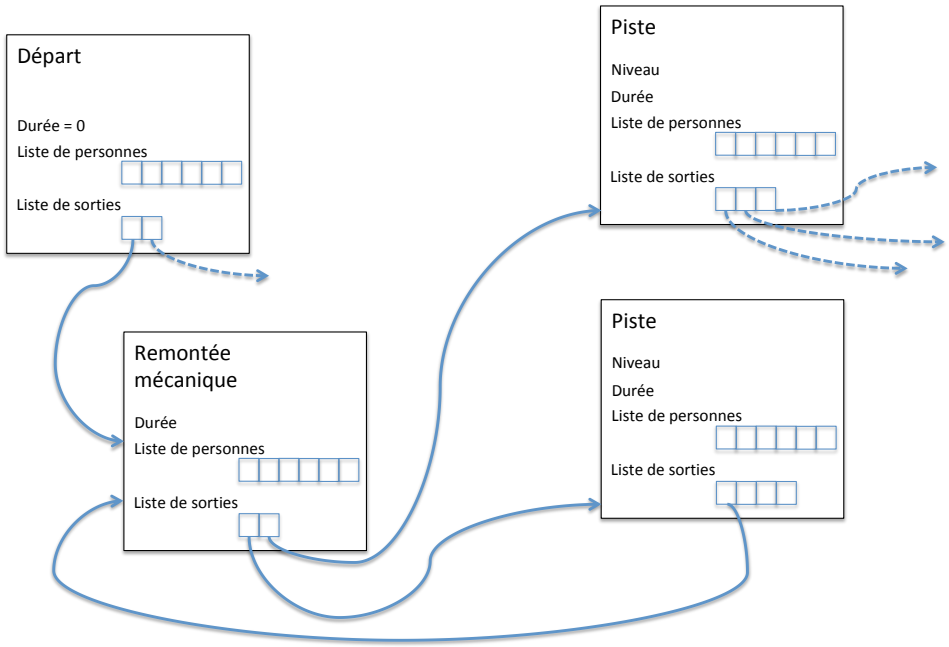
- Chaque personne possède un identifiant unique attribué automatiquement à la création de l'instance.
- Chaque personne a un `niveau` (entier entre 1 et 4) et un `forfait`. Ces deux données sont fournies en argument au constructeur.
- Chaque personne gère un entier `tpsDernierPassage`, initialisé à 0, qui permet de stocker le temps au moment de la dernière validation. La méthode `public void validation()` utilise la classe `Temps` pour mettre à jour `tpsDernierPassage` avec le temps courant.
- L'attribut `niveau` possède un accesseur. La classe possède aussi une méthode `public boolean estForfaitFini()` permettant de vérifier la validité du forfait.
- La méthode `getMateriel()` retourne une chaîne de caractères décrivant le matériel de la personne (`"ski"` ou `"surf"` dans le cadre de cet exercice).
- La méthode standard `toString()` donne l'`id` de la personne et son matériel.

Q 6.1 (6.5 pts) Donner le code des classes `Personne` et `Skieur`.

Exercice 7 – Gestion des pistes & remontées (12 pts)

Les remontées mécaniques, les pistes, etc peuvent toutes être représentées par une file de personnes. Les personnes rentrent dans la file, restent un moment puis sont transférées dans la file suivante (autre piste, autre remontée...).

Pour gérer la station, nous allons donc partir d'une classe `FilePersonnes` gérant une liste de `Personne` et une liste de sorties (de type `FilePersonnes`). L'idée est d'obtenir une architecture chaînée comme celle illustrée ci-dessous. Dans chaque `FilePersonnes`, les `Personne` vont passer un certain temps puis sortir dans l'une des autres `FilePersonnes` qui lui sont connectées jusqu'à expiration des forfaits. A chaque entrée dans une `FilePersonnes`, la `Personne` validera pour permettre une bonne gestion du temps. Cette définition de `FilePersonnes` permet de factoriser du code pour modéliser à la fois des pistes et des remontées mécaniques. Nous introduirons également une classe `Depart` permettant de regrouper toutes les personnes entrant dans la station.



Exemple : Une remontée mécanique (qui est une `FilePersonnes`) possède une liste de personnes. A la sortie, cette remontée est connectée à 2 pistes. Les personnnes vont donc soit aller sur la piste 1 soit sur la piste 2 (chaque piste étant une `FilePersonnes`).

Q 7.1 (2 pts) Donner le code de la classe abstraite `FilePersonnes` qui répond aux spécifications suivantes :

- Constructeur à un argument (la durée de la file) qui initialise tous les attributs. Les personnes et les sorties seront gérées par des `ArrayList` (mini-documentation à la fin).
- Méthodes d'ajout de personnes `public void ajouterPersonne(Personne p)` et d'ajout de sorties sur la structure `public void ajouterSortie(FilePersonne file)`. Attention, il faut penser à faire valider les personnes entrant dans la file.

— Un accesseur sur le nombre de personnes dans la file.

— Une méthode retournant le niveau de la file. Cette méthode sera implémentée dans les classes filles. Chaque piste à un niveau (entre 1 et 4) qui la rend accessible seulement aux personnes qui ont un niveau supérieur ou égal. Les remontées ont le niveau de la plus facile des sorties. Pour simplifier, nous ferons l'hypothèse que toutes les remontées possèdent une piste facile en sortie et nous règlerons simplement le niveau des remontées à 0.

— Enfin, chaque file possèdera une méthode de mise à jour `public void update()` qui sera détaillée dans la question suivante.

Q 7.2 (4 pts) Question difficile pouvant être gardée pour la fin. Mise à jour de la `FilePersonne`. Cette procédure n'est pas triviale et fait donc l'objet d'une question à part. Dans la méthode `update`, vous implémenterez les étapes suivantes :

— Elimination des personnes dont le forfait a expiré.

— Selection des personnes qui ont fini cette file (le temps de leur dernière validation + la durée de la file < temps courant) et orientation aléatoire de ces personnes sur les sorties éligibles (celles dont le niveau est compatible avec la personne). Le plus simple est de procéder avec une boucle `while` pour chaque personne à ré-orienter :

1. choix d'un indice de sortie aléatoire, initialisation des itérations à 0.
2. Tant que la sortie choisie est trop dure
   (a) choix d'un nouvel indice de sortie aléatoire
   (b) En cas de cul de sac (si nous ne trouvons pas de sortie admissible en 25 itérations par exemple), lever une RuntimeException avec un message ad'hoc.

Q 7.3 (2 pts) Donner le code de la classe `RemonteeMecanique` répondant aux spécifications suivantes :

— La remontée mécanique possède une capacité entière, initialisée à la création de l'objet, en plus de la durée.

— La méthode `update` a le même comportement que celui de la classe mère, mais elle ajoute un affichage donnant le nombre de personne sur la structure par rapport à la capacité.
Note : il s'agit d'un simple affichage, pas de test à prévoir.

— Comme expliqué précédemment, toutes les remontées mécaniques ont un niveau fixé à 0 pour plus de simplicité.

Q 7.4 (1 pt) Donner le code de la classe `Piste` qui étend également `FilePersonnes` et possède un niveau et une durée.

Q 7.5 (1 pt) Donner le code de la classe `Depart` qui étend `RemonteeMecanique` et a simplement une durée nulle (0) et un niveau 0. Cet héritage permet de gérer la capacité de la station.
Donner le code minimum de la classe sans ajouter de code inutile.

Q 7.6 (OPT, 1 pt) Donner le code permettant de stocker et d'accéder au nombre total de personnes étant passées par le départ

Exercice 8 – Classe de test (6 pts)

Construire une station composée d'un départ, d'une remontée et de deux pistes (niveau 1 et 2) revenant en bas de la remontée. Ajouter 200 personnes au départ (20% de surfeur et 80% de skieurs) ayant des forfaits jour (50% de la population) ou demi-journée (l'autre moitié de la population). Les personnes ont un niveau aléatoire tiré entre 1 et 5. Durant 300 itérations temporelles, faire évoluer la micro-station.

Rappel de documentation : `ArrayList<Object>`

Il faut ajouter la commande `import java.util.ArrayList;` en début de fichier pour utiliser les `ArrayList`.

— Instanciation : `ArrayList<Object> a = new ArrayList<Object>();`

— `void add(Object o)` : ajouter un élément à la fin

— `Object get(int i)` : accesseur à l'item `i`

— `Object remove(int i)` : retirer l'élément et renvoyer l'élément à la position `i`

— `int size()` : retourner la taille de la liste

— `boolean contains(Object o)` retourne `true` si `o` existe dans la liste. Le test d'existence est réalisé en utilisant `equals` de `o`.