

2i002 – Examen final noté sur 60 pts

Durée : 2 heures

*Aucun document autorisé
– Barème 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()**.

1 point pour cette question ?

```
1 public Lapin(String nom) {
2     this(nom,0);
3 }
```

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.

-0.5 par faute (2 fautes possibles par ligne)

Résultat attendu :

```
lig14 --> true true
lig15 --> false false
lig16 --> true true
lig17 --> false false
```

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**.

1 point pour le **.equals** sur les string
1 point pour le reste (0 à la moindre erreur)

```
1 public boolean equals(Object obj) {
2     if(obj == null) return false;
```

```

3    if(this.getClass() != obj.getClass()) return false;
4    Lapin l2 = (Lapin) obj;
5    return (age == lapin.age)&&(nom.equals(l2.nom));
6 }

```

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.

0 à la moindre erreur

Pour coller à la question 1.6, il faut une solution non statique.

```

1 public Lapin reproduire(Lapin lapin1) {

```

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**.

1 pt si tout est ok. Enlever 0.5 si pas de traitement du cas null.
0 si pas de return ou d'autres fautes

```

1    public Lapin reproduire(Lapin lapin1) {
2        if (lapin1 == null)
3            return null;
4        if ((lapin1 != this) && (lapin1.age == age) && (age > 0) &&
5            (nom.charAt(0) == lapin1.nom.charAt(0)))
6            return new Lapin(nom + lapin1.nom, 0);
7        return null;
8    }
9

```

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**.

<pre> 1 Lapin[] tab = new Lapin[10]; 2 tab[0] = new Lapin("toto", 1); 3 tab[1] = new Lapin("titi", 1); 4 tab[2] = tab[0].reproduire(tab[1]); 5 tab[2].vieillir(); 6 tab[3] = tab[2].reproduire(tab[0]); 7 tab[4] = tab[3]; 8 for(int i=0; i<5; i++) tab[i].vieillir(); 9 tab[5] = tab[3].reproduire(tab[4]); </pre>	<pre> 10 Lapin parent1 = tab[0]; 11 Lapin parent2 = tab[1]; 12 tab[0] = new Lapin("lapinou", 3); 13 tab[1] = new Lapin("laplap", 4); 14 for(int i=5; i<10; i++) 15 tab[i] = parent1.reproduire(parent2); 16 tab[6] = null; 17 tab[8] = null; </pre>
--	--

1pt ⇒ 4 instances dans la première colonne
1pt pour + 7 instances dans la colonne 2
1 pt pour -2 instances tjs existantes

Colonne 1 : instances créées :

- 2 instances lig2 et lig3
- 1 instance lig4 : reproduction marche
- 1 instance lig6 : reproduction marche
- pas d'instance lig9 : pas de reproduction car même référence

Soit en tout : 4 instances créées et en mémoire après la ligne 9.

Colonne 1 et 2 : 11 instances créées

Nombre d'instances restant à la fin : 9 instances.

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.

1pt pour la détection de l'erreur et 1pt pour sa correction

Il y a une erreur de compilation :

- `in` n'est pas visible dans `finally` car il faut déclarer `FileInputStream in` avant le `try` :

```

$ javac TestLectureFichier.java
TestLectureFichier.java:21: error: cannot find symbol
in.close();
~
  symbol:   variable in
  location: class TestLectureFichier
1 error

```

Remarque : pas d'erreur pour `(c=='!')` car les types sont compatibles.

La classe sans erreur :

```

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         FileInputStream in = null;
7         try {
8             in = new FileInputStream(f);

```

```

9          int c = in.read();
10         while (c != -1) {
11             if (c == (int)')')
12                 v +=1;
13             if (v > 2)
14                 throw new IOException("Trop de !");
15             c = in.read();
16         }
17     }
18     catch (Exception e) {
19         System.err.println("\nErreur" + e);
20     }
21     finally {
22         if (in != null) {
23             in.close();
24         }
25     }
26     System.out.println("v=" + v);
27 }
28 }

```

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

1pt pour le résultat correct de la première ligne de l'exécution avec le bon nom d'exception 1pt pour la 2e ligne correcte (valeur de v).

Une exception est levée :

```

$ java TestLectureFichier
Erreur java.io.IOException: Trop de !
v= 3

```

L'IOException est trappée par le catch Exception car IOException est une sous-classe, donc tout est ok.

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.

1.5pt pour la détection du problème et 0.5pt pour sa correction. Il faut bien dire que c'est une exception `NullPointerException` qui est levée là où on s'attendrait à une `IOException`.

On a une exception `NullPointerException` :

```

$ java TestLectureFichier
Erreur java.io.FileNotFoundException: data.txt (Aucun fichier ou dossier de ce type)
Exception in thread "main" java.lang.NullPointerException
at TestLectureFichier.main(TestLectureFichier.java:22)

```

En effet, le `close()` ne fonctionne pas si le fichier n'a pas été trouvé (et donc ouvert). Il faut rajouter un test de nullité pour `in` :

```

1         finally {
2             if (in != null) {
3                 in.close();
4             }
5         }

```

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

1pt pour l'explication correcte.

L'instruction `in.close()` est susceptible de lever une `IOException`, il est donc nécessaire soit de la trappée, soit de préciser que la fonction `main` peut lever cette exception.

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 }

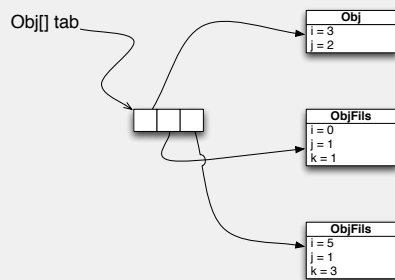
16 public class ObjFils extends Obj{
17     private int k;
18     public ObjFils(int i, int j) {
19         super(i, j);
20         this.k = i+j+(int) Math.random();
21     }
22     public ObjFils(int m) {
23         this(m, m+1);
24     }
25     public int getResult(){
26         System.out.println("res_("+fils)");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

0.5 point pour la représentation du tableau en tant qu'objet (avec 3 cases et une variable `tab` clairement séparée)
0.5 point pour les 3 case avec les bons types d'objet

Assez trivial sans les valeurs numériques... Mais je veux voir les types de la variable et des instances.



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

1 pt : `(int) Math.random()` vaut toujours 0

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

0.5 pt : affichage des méthodes (respect du nb d'appel à `maj`)
1 pt sélection des méthodes `res`
1.5 pt : pour le calcul de 17

```

1 maj
2 maj
3 maj
4 res
5 res ( fils )
6 res ( fils )
7 17
  
```

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 ?

Non, ne pas confondre `super.` et `super()`

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));
  
```

0 si faux, 2 si juste :)

```
1 ObjFils of = new ObjFils(1); // 1, 2, 3
2 of.maj(new Obj(1,1)); // 2, 2, 3
3 of.maj(new ObjFils(1,1)); // Resultat attendu : 3, 2, 5
```

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-)

0 si mauvaise explication

Ca donne une récursion infinie (plantage sur StackOverflow)

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).

Non, aucun changement, la méthode n'est jamais sélectionnée!

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`.

1 pt pour l'instance private static
0.5 pour la signature de classe + constructeur
1.5 pt pour les deux méthodes avec délégation sur t
Si BONNE gestion du temps mais sans singleton ⇒ 1.5 pt

```
1 package exam2018;
2
3 public class Temps {
4
5     private static Temps t = new Temps();
6     private int currentTime;
7     private Temps() {
8         currentTime = 0;
9     }
10    public static void avancer() {
```

```
11         t.currentTime ++;  
12     }  
13     public static int getTemps() {  
14         return t.currentTime;  
15     }  
16 }
```

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**.

1 pt CST
0.5 constructeur
1 signature des factory static
1 gestion du temps

```
1 public class Forfait {  
2     private static final int DEMI_JOURNEE = 240;  
3     private static final int JOURNEE = 480;  
4     private static final int HEURE = 60;  
5  
6     private int tmax;  
7  
8     private Forfait(int tmax) {  
9         this.tmax = tmax;  
10    }  
11    public boolean estFini() {  
12        return Temps.getTemps() > tmax;  
13    }  
14    public static Forfait buildForfaitDJ() {  
15        return new Forfait(Temps.getTemps() + DEMI_JOURNEE);  
16    }  
17    public static Forfait buildForfaitJ() {  
18        return new Forfait(Temps.getTemps() + JOURNEE);  
19    }  
20    public static Forfait buildForfaitH() {  
21        return new Forfait(Temps.getTemps() + HEURE);  
22    }  
23 }
```

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

```
1     Forfait f = Forfait.buildForfaitDJ();
```

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**.

1 cpt static
1.5 constructeur + validation + validité forfait
2 pts méthode abstraite + appel de cette méthode dans le toString
1.5 pour Skieur (0.5 appel super + 1 getMateriel)

```

1 public abstract class Personne { // classe abstraite
2     private static int cpt = 0;
3     private int id;
4     private int niveau;
5     private int tpsDernierPassage;
6
7     private Forfait forfait;
8     public Personne(int niveau, Forfait forfait) {
9         id = cpt++;
10        this.niveau = niveau;
11        this.forfait = forfait;
12        tpsDernierPassage = 0;
13    }
14    public boolean estForfaitFini() {
15        return forfait.estFini();
16    }
17    public void validation() {
18        tpsDernierPassage = Temps.getTemps();
19    }
20    public int getTpsDernierPassage() {
21        return tpsDernierPassage;
22    }
23    public int getNiveau() {
24        return niveau;
25    }
26    public String toString() {
27        return "id: " + id + " (" + getMateriel() + ")";
28    }
29    public abstract String getMateriel(); // methode abstraite. Eventuellement,
30        // attribut sup + réglage par défaut dans le constructeur de la classe fille
31
32 }
33 public class Skieur extends Personne {
34     public Skieur(int niveau, Forfait forfait) {
35         super(niveau, forfait);
36     }
37
38     @Override
39     public String getMateriel() {
40         return "ski";
41     }
42 }

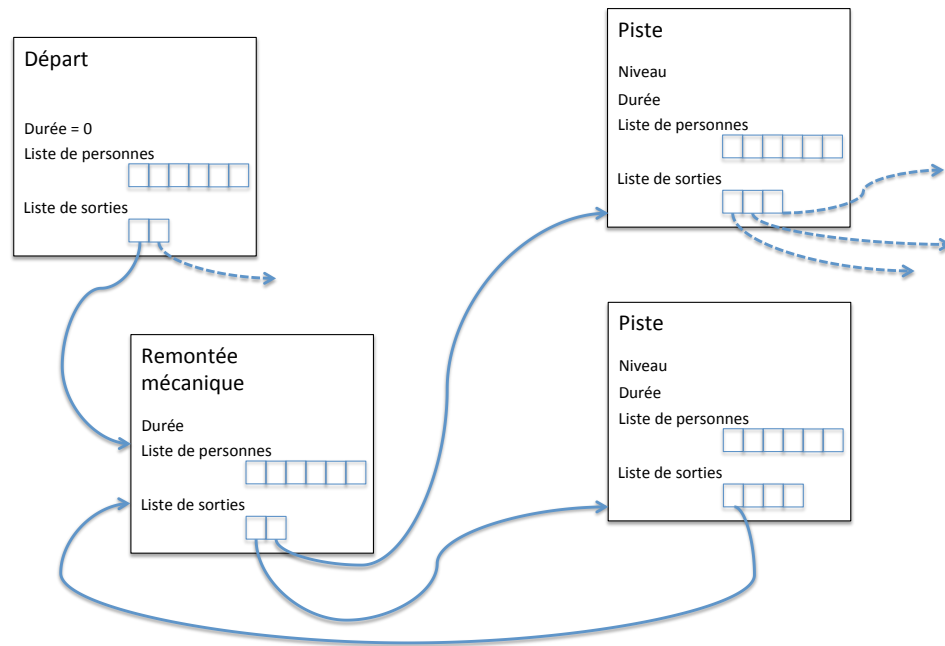
```

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 **Départ** 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 personnes 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 réglerons 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.

0.5 constr/attribut

0.5 ajout

0.5 size

0.5 getNiveau

UPDATE

1pt enlever les personnes dont le forfait est fini

1pt détection des personnes qui ont fini le "tour" sur la file

2 pt affectation des personnes à une nouvelle file aléatoirement

pénalisation de 1 pt pour ceux qui enlèvent mal les personnes en cours d'itération sur la file

```

1 import java.util.*;
2
3 import javax.management.RuntimeErrorException;
4
5 public abstract class FilePersonne {
6
7     private ArrayList<Personne> personnes;
8     private ArrayList<FilePersonne> sorties;
9     private int duree;
10
11
12     public FilePersonne(int duree) {
13         this.duree = duree;
14         sorties = new ArrayList<FilePersonne>();
15         personnes = new ArrayList<Personne>();
16     }
17
18     public void ajouterPersonne(Personne p) {
19         personnes.add(p);
20         p.validation();
21     }
22
23     public void ajouterSortie(FilePersonne file) {
24         sorties.add(file);
25     }
26
27     // public Personne enlever() {
28     //     return personnes.remove(0);
29     // }
30
31     public int size() {
32         return personnes.size();
33     }
34
35     public abstract int getNiveau();
36
37     public void update() {
38         ArrayList<Personne> toRemove = new ArrayList<Personne>();
39         ArrayList<Personne> toMove = new ArrayList<Personne>();
40         for(Personne p: personnes) {
41             // interdiction d'utiliser le else
42             if(p.estForfaitFini()) {
43                 toRemove.add(p);
44                 System.out.println(p + "sort_de_la_station_à_" + Temps.getTemps());
45                 continue;
46             }
47             if(p.getTpsDernierPassage()+duree<Temps.getTemps()) {
48                 toMove.add(p);
49                 continue;
50             }
51         }
52         for(Personne p: toRemove)
53             personnes.remove(p);
54         for(Personne p: toMove) {
55             personnes.remove(p);
56             int indexRand = (int)(sorties.size() * Math.random());
57             int cptInf=0;
58             while(sorties.get(indexRand).getNiveau()>p.getNiveau() && cptInf < 25) {
59                 indexRand = (int)(sorties.size() * Math.random());
60                 cptInf ++;

```

```

61         }
62         if(cptInf == 25)
63             throw new RuntimeException("Personne prise dans un cul de sac!!!"+ p);
64         sorties.get(indexRand).ajouterPersonne(p);
65     }
66
67
68
69 }
70
71
72
73 }

```

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.

0.5 constructeur

1 update mis à jour avec appel a super.update

0.5 getniveau

```

1 package exam2018;
2
3 public class RemonteeMecanique extends FilePersonne{
4
5     private int capacite;
6     public RemonteeMecanique(int duree, int capacite) {
7         super(duree);
8         this.capacite = capacite;
9     }
10
11     public void update() {
12         System.out.println("Il y a actuellement "+size()+ " personnes sur la remontée");
13         super.update();
14     }
15
16     @Override
17     public int getNiveau() {
18         return 0;
19     }
20 }

```

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

```

1 public class Piste extends FilePersonne{
2
3     private int niveau;
4     public Piste(int duree, int niveau) {
5         super(duree);
6         this.niveau = niveau;
7     }
8
9     public int getNiveau() {
10         return niveau;
11     }
12
13 }

```

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.

```

1 public class Depart extends RemonteeMecanique{
2
3     public Depart(int capacite) {
4         super(0, capacite);
5     }
6
7 }
```

Pénalisation en cas de redéfinition de `getNiveau`

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

- Attribut `cpt`
- Surcharge du `ajouterPersonne` + incrément `cpt`
- Accesseur `cpt`

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.

2 pts pour la construction de la liste chaînée

2.5 pts pour la génération aléatoire de la population : gen forfait puis niveau aléatoire

1.5 pour la boucle temporelle (avec avancement du Temps)

⇒ 0.5 si tout faux mais bonne volonté

```

1 public static void main(String[] args) {
2
3     // Construction de la station
4     Piste bleue = new Piste(10, 2);
5     Piste verte = new Piste(12, 1);
6
7
8     Depart dep = new Depart(1000);
9     RemonteeMecanique telesiegel = new RemonteeMecanique(8, 300);
10    // Liens entre les elements
11    dep.ajouterSortie(telesiegel);
12    telesiegel.ajouterSortie(verte);
13    telesiegel.ajouterSortie(bleue);
14    verte.ajouterSortie(telesiegel);
15    bleue.ajouterSortie(telesiegel);
16
17    // ajouter tout le monde dans la station
18
19    int nbPersonnes = 200;
20    for(int i=0; i<nbPersonnes; i++) {
21        Forfait f;
22        if(Math.random() > 0.5)
23            f=Forfait.buildForfaitDJ();
24        else
25            f=Forfait.buildForfaitJ();
26    }
```

```
27         if(Math.random() > 0.2)
28             dep.ajouterPersonne(new Skieur((int) (Math.random() * 4)+1, f));
29         else
30             dep.ajouterPersonne(new Surfeur((int) (Math.random() * 4)+1, f));
31     }
32
33     for(int iter = 0; iter<200; iter++) {
34         Temps.avancer();
35         dep.update();
36         telesiegel.update();
37         verte.update();
38         bleue.update();
39         System.out.println("Time: "+Temps.getTemps());
40     }
41 }
```

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`.