2i002 – Examen final noté sur 60 pts

Durée : 2 heures

Questions de cours (30 pts)

Exercice 1 – Vocabulaire & syntaxe de base JAVA (5 pts)

Soit le code comprenant la classe `MaClasse` et un `main` :

```
1 public class MaClasse{
2   String name;
3   int i;
4   MaClasse(int i, String name){
5     this.i = i;
6     this.name = name;
7   }
8   void affichage(){
9     System.out.println("nom: "+name+" i="+i);
10  }
11  static int getNbInstances(){
12   // à compléter dans les questions suivantes
13  }
14 }
```

```
15 public static void main(String[] args){
16
17 MaClasse m = new MaClasse(2, "toto");
18 MaClasse m2 = new MaClasse(4, "titi");
19 MaClasse m3 = m;
20
21 m.affichage();
22 System.out.println(MaClasse.
       getNbInstances());
23 }
```

Q 1.1 (1 pt) Le code est fonctionnel mais il manque les déclarations de droit d'accès sur les méthodes et attributs. Indiquer, pour chaque ligne où c'est nécessaire, les modifieurs (`public/protected/private`) à ajouter pour obtenir un code sécurisé.

Barème : -0.5 par oubli/ajout non pertinent

- private sur les attributs
- public sur les méthode

Q 1.2 (2 pts) Pour chacune des propositions suivantes, écrire le numéro de la proposition, puis dire si la proposition est vrai ou fausse (dans ce dernier cas, justifier en une phrase)

1. `m3` est une *variable* de type `MaClasse`.
2. `m2` est une *instance* de type `MaClasse`.
3. `m2` et `m3` référencent la même instance.
4. Aux lignes 5 et 6, le `this` est optionnel : si on l'enlève le comportement du programme reste le même.
5. L'instruction `new` permet de créer une nouvelle *instance* de type `MaClasse`.
6. `affichage` est une *méthode d'instance* de `MaClasse`.
7. Dans la méthode `getNbInstances`, je peux ajouter l'instruction suivante :

```
System.out.println("la méthode a été invoquée depuis l'objet nommé: "+name);
```

Barème : -0.5 par faute, -0.25 par NON non justifié

1. OUI
2. NON : m2 est une variable
3. NON : 2 new, 2 instances
4. NON : confusion entre argument et attribut si oubli du this
5. OUI
6. OUI
7. NON : la méthode est static, elle n'a pas accès aux attributs des instances

Q 1.3 (2 pts) Donner le code de la méthode `getNbInstances` ainsi que les attributs et les lignes du constructeur nécessaires au bon fonctionnement de la méthode (création d'un compteur d'instance et incrément dans le constructeur).

Barème : 1/2 pt pour static, 1/2 pt pour init à 0 dans les attributs, 1 pt pour le reste

```
1
2 private static int cpt = 0;
3
4 public MaClasse(int i, String s){
5   cpt++;
6 }
7
8 public static int getNbInstances(){ return cpt;}
```

Exercice 2 – Instances et références (8 pts)

Soit les classes suivantes `A` et `B` et le `main` associé :

```
1 public class A{
2   private String str;
3   public A(String str){
4     this.str = str;
5   }
6   public A clone(){
7     return new A(str);
8   }
9   public String getStr(){
10    return str;
11  }
```

```
14 public static void main(String[] args){
15   A a1 = new A("toto");
16   A a2 = a1.clone();
17   A a3 = a1;
18   String str = a1.getStr();
19 }
```

Q 2.1 (2 pts) Combien y a-t-il d'instances de la classe `A` et combien d'instances de la classe `String` lorsque l'exécution du `main` se termine (ligne 19) ? Expliquer brièvement.

Barème : 1 pt pour les A, 1 pour les String
Réponse : 2 et 1

Q 2.2 (2 pts) Redéfinition de `equals`. Donner le code de la méthode standard `boolean equals(Object o)` à insérer dans la classe `A` pour obtenir le même comportement entre les deux lignes de code suivantes :

```
19 // à la suite du main précédent
20 a1.equals(a2);
21 a1.equals(a3);
```

- -0.5 par if oubilé
- -1 si pas de cast
- -0.5 pour == au lieu de .equals sur la string
- Pas de pénalisation pour les tests de null sur la string

```
1 boolean equals(Object o){
2   if(o == null) return false;
3   if(o == this) return true; // OPT
4   if(o.getClass() != getClass()) return false;
5   A o2 = (A) o;
6   return str.equals(o2.str);
7 }
```

Q 2.3 (1 pt) Pourquoi est-il essentiel d'utiliser le nom `equals` et pas n'importe quel autre nom de fonction (eg `egalite`) ?

0.5 Pour les usages implicites (par exemple dans le contains de ArrayList)
0.5 Pour les autres codeurs qui reprendront le code

Q 2.4 (1 pt) Pourquoi est-il essentiel de déclarer l'argument de `equals` de type `Object` ?

Pour éviter les ambiguités dans les sélections de méthode et les appels inopinés à equals de Object

Q 2.5 (2 pts) Références et instances

```
19 // à la suite du main précédent
20 boolean b1 = a1 == a2;
21 boolean b2 = a1.getStr() == a2.getStr();
```

1. Est-ce que la syntaxe des lignes de code précédentes est correcte ? Sinon, proposer une correction.
2. Que vallent `b1` et `b2` ? Justifier la réponse en décrivant une représentation de l'état de la mémoire.

Barème :

— 1 pour la question 1

— 1 pour la question 2 : avec justif

La syntaxe est correcte (il faut qu'ils comprennent que == est un opérateur classique qui renvoie un booléen pouvant être stocké de manière classique)
b1 = false, b2 = true
Les objets sont différents (2 instances avec l'usage du clone)... Mais la chaine de caractères est la même pour les deux objets (passage de la référence dans le clonage de A).

Exercice 3 – Recherche de minimum et exception (5 pts)

Dans la classe `Toolbox`, nous souhaitons implémenter une méthode de classe pour rechercher la valeur minimale dans un tableau de réels. Nous allons procéder en exploitant le fonctionnement des exceptions pour combler les trous (①, ②, ③, ④).

```
1  public class Toolbox {
2      public static double mini(double[] liste) {
3          int i_min = 0;
4          int cpt = 1;
5          ①
6          while(true) {
7              ②
8              test(liste[i_min] , liste[cpt]);
9              ③
10              cpt++;
11         }
12         ④
13     }
14     public static void test(double a, double b) {
15         if(a>b) throw new MinException("a supérieur à b");
16     }
17     public static void main(String[] args) {
18         System.out.println(mini(new double[]{2., 5., 1., 6., 7.}));
19     }
20 }
```

Q 3.1 (1 pt) Donner le code de la classe `MinException` pour l'usage prévu à la ligne 15

```
1 public class MinException extends Exception {
2     public MinException(String message) {
3         super(message);
4     }
5 }
```

Q 3.2 (1 pt) La méthode `test` ne compile pas en l'état : proposer une correction.

throws MinException

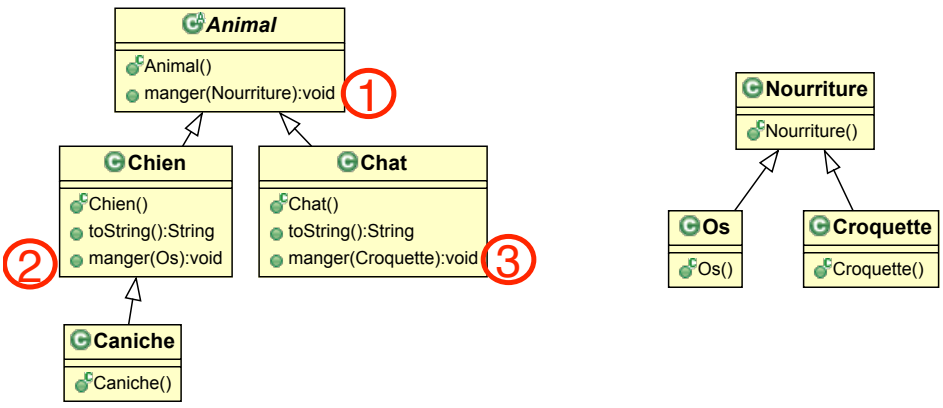
Q 3.3 (3 pts) Compléter la méthode `mini` pour qu'elle retourne le minimum de la liste fournie en argument.
Vous ne pouvez ajouter du code **que** dans les cases ①, ②, ③, ④
Vous ne pouvez pas déclarer de nouvelles variables / compteurs...
Vous ne pouvez pas ajouter d'instruction `if` ni `for`.
Vous vous rappellerez que la sortie d'un tableau provoque une exception `IndexOutOfBoundsException`.

```
1 public class Toolbox {
2     public static double mini(double[] liste) {
3         int i = 0;
4         int cpt = 1;
5         try {
6             while(true) {
7                 try {
8                     test(liste[i] , liste[cpt]);
9                 }catch(MinException e) {
10                    i = cpt;
11                }
12                cpt++;
13            }
14        }catch(IndexOutOfBoundsException e) {
15            return liste[i];
16        }
17    }
18
19    public static void test(double a, double b) throws MinException {
20        if(a>b) throw new MinException("a>b");
21    }
22
23    public static void main(String[] args) {
24        System.out.println(mini(new double[]{2., 5., 1., 6., 7.}));
25    }
26 }
```

Exercice 4 – Des chats, des chiens, des conversions, des méthodes à sélectionner (7 pts)

Nous nous plaçons dans le cadre défini par le schéma UML suivant (`Animal` étant en italique pour souligner le caractère abstrait de la classe) :

```
1 public static void main(String[] args) {
2     Animal  a1   = new Chien();
3     Animal  a2   = (Animal) new Caniche();
4     Animal  a3   = (Chien) new Caniche();
5     Animal  a4   = new Animal();
6     Animal  a5   = new Chat();
7     Chat    cat1 = new Chat();
8     Chat    cat2 = (Chat) a1;
9     Chat    cat3 = (Chat) a5;
10    Chien   dog1 = new Chat();
11    Chien   dog2 = new Caniche();
12    Chien   dog3 = (Chien) a1;
13    Caniche dog4 = new Chien();
14    Caniche dog5 = (Caniche) a1;
15    Caniche dog6 = (Chien) a1;
16 }
```



Q 4.1 (2 pts) Parmi les lignes précédentes, lesquelles posent problème à la compilation et lesquelles posent problème à l'exécution ?

-0.5 par erreur ou oubli
Compilation : 5, 10, 13, 15
Exécution : 8, 14

Q 4.2 (2.5 pts) Donner le code **minimal** de la classe `Refuge`, héritant de `ArrayList<Animal>` permettant de :

— Construire le refuge sans argument.

— Ajouter des animaux dans le refuge, en enlever.

— Compter les animaux, compter les chiens (les caniches étant évidemment des chiens).

Note : attention à ne pas donner de code en trop !

Besoin de presque rien !!!

— Pas de pénalisation si on met le constructeur

— -1 : Pénalisation en cas d'ajout de méthode add etc...

— -1 si syntaxe incorrecte

— 1 pour compterChiens

— 0.5 si mauvaise gestion instanceof avec Chien/Caniche

```
1 import java.util.ArrayList;
2
3 public class Refuge extends ArrayList<Animal> {
4
5     public Refuge() { // facultatif mais on ne pénalise pas sur le constructeur
6         super();
7     }
8
9     public int comptetChien() {
10        int cpt=0;
11        for(Animal a :this)
12            if(a instanceof Chien) // pas de getClass !!!
13                cpt ++;
14        return cpt;
15    }
16
17 }
```

Q 4.3 (2.5 pts) Le code suivant appelle différentes méthodes `manger`. Pour les 4 lignes de codes 12 à 15, pour chacune des 5 instructions de la ligne, indiquer la méthode `manger` invoquée (①, ② ou ③). En cas de problème de compilation/exécution mettre une croix.
Pour plus de clarté, vous présenterez la solution sous la forme d'un tableau 4 lignes x 5 colonnes.

```
1 Animal a1  = new Chien();
2 Animal a2  = new Chat();
3 Chat   cat = new Chat();
4 Chien  dog = new Chien();
5
6 Nourriture n1 = new Nourriture();
7 Nourriture n2 = new Os();
8 Nourriture n3 = new Croquette();
9 Os o1 = new Os();
10 Croquette c1 = new Croquette();
11
12 a1.manger(n1);  a1.manger(n2);  a1.manger(n3);  a1.manger(o1);  a1.manger(c1);
13 a2.manger(n1);  a2.manger(n2);  a2.manger(n3);  a2.manger(o1);  a2.manger(c1);
14 cat.manger(n1); cat.manger(n2); cat.manger(n3); cat.manger(o1); cat.manger(c1);
15 dog.manger(n1); dog.manger(n2); dog.manger(n3); dog.manger(o1); dog.manger(c1);
```

-0.5 par erreur

```
1  1  1  1  1
1  1  1  1  1
1  1  1  1  3
1  1  1  2  1
```

Exercice 5 – Programme mystère (de Noël) (5 pts)

Nous disposons du programme principal suivant :

```
ArbreDeNoel a1 = new Epicea();
ArbreDeNoel a2 = new Nordmann(1.5);
a1.add(new Boule("rouge"));
a1.add(new Guirlande(2));
a2.add(new BouleANeige());
a1.add(new Guirlande(3.5));
Decoration d = new Boule("bleue");
a1.add(d);
a2.add(d.dupliquer());
a1.add(d.dupliquer());
```

A l'issue de l'exécution de la dernière ligne, le programme a planté avec le message suivant :

```
Exception in thread "main" TropDeDecorationException : l'arbre␣est␣trop␣chargé!
```

Q 5.1 (5 pts) Donner la hiérarchie des classes en présence dans ce programme et les signatures complètes –et localisations– de toutes les méthodes utilisées.
Indice : réfléchir à la possibilité de méthodes abstraites.

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 6 – 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 et correspondant à des minutes), 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 :

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

Q 6.1 (3 pts) Donner le code de la classe `Temps`.

```
package exam2018;

public class Temps {

    private static Temps t = new Temps();
    private int currentTime;
    private Temps() {
        currentTime = 0;
    }
    public static void avancer() {
        t.currentTime ++;
    }
```

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

Exercice 7 – 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 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 7.1 (3.5 pts) Donner le code de la classe `Forfait`.

```
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 7.2 (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 8 – Gestion des personnes (5 pts)

Les **personnes** seront ici déclinées en **skieurs** ou **surfeurs**. 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 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.

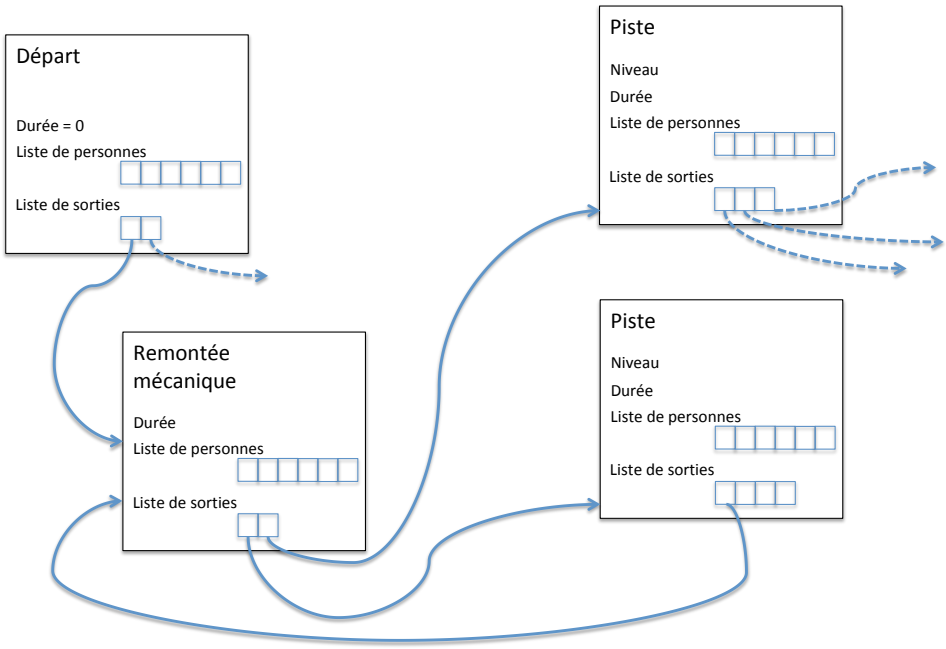
Q 8.1 (5 pts) Donner le code des classes `Personne` et `Skieur`.

```
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 claasse 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 9 – 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.



Q 9.1 (3 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 9.2 (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.

```
1 package exam2018;
2
3 import java.util.*;
4
5 import javax.management.RuntimeErrorException;
6
```

```java
public abstract class FilePersonne {

    private ArrayList<Personne> personnes;
    private ArrayList<FilePersonne> sorties;
    private int duree;


    public FilePersonne(int duree) {
        this.duree = duree;
        sorties = new ArrayList<FilePersonne>();
        personnes = new ArrayList<Personne>();

    }

    public void ajouterPersonne(Personne p) {
        personnes.add(p);
        p.validation();
    }

    public void ajouterSortie(FilePersonne file) {
        sorties.add(file);
    }
//    public Personne enlever() {
//        return personnes.remove(0);
//    }

    public int size() {
        return personnes.size();
    }

    public abstract int getNiveau();

    public void update() {
        ArrayList<Personne> toRemove = new ArrayList<Personne>();
        ArrayList<Personne> toMove = new ArrayList<Personne>();
        for(Personne p: personnes) {
            // interdiction d'utiliser le else
            if(p.estForfaitFini()) {
                toRemove.add(p);
                System.out.println(p + " sort de la station à "+Temps.getTemps());
                continue;
            }
            if(p.getTpsDernierPassage()+duree<Temps.getTemps()) {
                toMove.add(p);
                continue;
            }
        }
        for(Personne p:toRemove)
            personnes.remove(p);
        for(Personne p: toMove) {
            personnes.remove(p);
            int indexRand = (int)(sorties.size() * Math.random());
            int cptInf=0;
            while(sorties.get(indexRand).getNiveau()>p.getNiveau() && cptInf < 25) {
                indexRand = (int)(sorties.size() * Math.random());
                cptInf ++;
            }
            if(cptInf == 25)
                throw new RuntimeException("Personne prise dans un cul de sac !!! "+ p);
            sorties.get(indexRand).ajouterPersonne(p);
        }



    }


}
```

Q 9.3 (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.

```
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 9.4 (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 9.5 (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 9.6 (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 10 – 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.

```
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 telesiege1 = new RemonteeMecanique(8, 300);
10       // Liens entre les elements
11       dep.ajouterSortie(telesiege1);
12       telesiege1.ajouterSortie(verte);
13       telesiege1.ajouterSortie(bleue);
14       verte.ajouterSortie(telesiege1);
15       bleue.ajouterSortie(telesiege1);
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            telesiege1.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`.