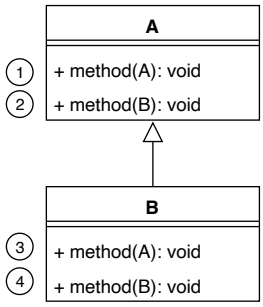
LU2IN002 – 2019-2020 – Examen Session 1

Janvier 2020 – Durée : 2 heures

Aucun document, pas de calculatrice. Le barème (sur 60) est donné à titre indicatif.

Exercice 1 (6.5pts) – Combien d'instances, quelle méthode ?




```
1  public static void
2       main(String[] args){
3    A[] tab = new A[5];
4    for(int i=0; i<4; i++){
5      if(i==0)
6        tab[i] = new A();
7      else
8        tab[i] = new B();
9    }
10   tab[1] = tab[0];
11   A a = naw A();
12   tab[2] = a;
13 }
```

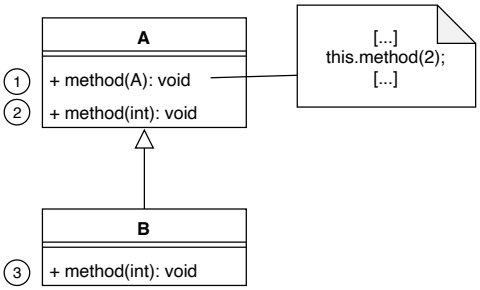
```
1 public static void
2       main(String[] args){
3   A a = new A(); B b = new B(); A ab = new B();
4   ab.method(a); ab.method(b); ab.method(ab);
5   a.method(a); b.method(a); ab.method(a);
6 }
```

Q 1.1 (2pts) Dans le `main` de la colonne du milieu, combien d'instances de A et de B ont été créées au total ? Donner les 2 chiffres. Combien en reste-t-il à l'issue de l'exécution du programme ?

Q 1.2 (1.5pt) Dans le `main` de droite à la **ligne 4**. Quelles sont les méthodes pré-sélectionnées par le **compilateur** ? Quelles sont les méthodes exécutées par la **JVM** ? Donner simplement 2 séries de 3 numéros par rapport à la numérotation proposée sur le diagramme UML.

Q 1.3 (1.5pts) Dans le `main` de droite à la **ligne 4**, **dans le cas où la méthode ② n'existe plus**. Répondre à la même question que précédemment.

Q 1.4 (1.5pt) En considérant la nouvelle architecture ci-contre et le `main` de droite précédent à la **ligne 5**. Quelle méthode est invoquée à l'intérieur de ① dans les 3 appels ?



Exercice 2 (13pts) – Encore des animaux : composition, clonage et égalité

Soit une classe `Vecteur` très basique et une classe `Animal`.

```
1 public class Vecteur {
2   public final double x,y;
3   public Vecteur(double x, double y) {
4     this.x = x; this.y = y;
5   }
6   public Vecteur() {
7     this.x = Math.random()*50;
8     this.y = Math.random()*50;
9   }
10 }
```

```
1 import java.util.ArrayList;
2 public class Animal {
3   public final int id;
4   private static int cpt = 0;
5   private Vecteur position;
6   private final static ArrayList<Animal> all =
        new ArrayList<Animal>();
7
8   public Animal(Vecteur position) {
9     id = cpt; cpt++;
10    this.position = position;
11    all.add(this);
12  }
13 }
```

Q 2.1 (0.5pt) Est-il raisonnable d'avoir déclaré certains attributs `public` ?

Q 2.2 (0.5pt) La déclaration `final` sur `all` est-elle un problème pour ajouter et enlever des éléments de la liste ? Faut-il la retirer ?

Q 2.3 (1pt) Dans la classe `Vecteur`, ajouter une méthode d'addition de vecteurs. Vous noterez que la

déclaration des attributs impose naturellement une unique solution possible pour l'implémentation de l'addition et la signature de la méthode.

Q 2.4 (2pts) Dans la classe `Animal`, à quoi sert l'attribut `all` ? Ajouter un accesseur `getAnimal` sur l'élément `ind` de la liste : cet accesseur retournera l'`Animal` correspondant s'il existe et `null` sinon.

Q 2.5 (3pts) Donner le code des redéfinitions standards des méthodes `equals` et `clone` pour `Animal`. Réfléchir à la bonne manière de gérer `id` et `all`, ajouter du code dans `Vecteur` si nécessaire et bien faire attention aux problèmes de composition. Nous sommes particulièrement intéressés à détecter le fait que deux animaux se trouvent à la même position.

Q 2.6 (1.5pt) Après avoir proprement introduit le `equals` dans `Animal` et en imaginant deux classes filles `Koala` et `Panda` pour lesquelles la méthode `equals` n'est pas redéfinie. Quelles seraient les sorties associées au code suivant :

```
Vecteur pos = new Vecteur(1, 2);
Koala k   = new Koala(pos); Panda p   = new Panda(pos);
Animal ak = new Koala(pos); Animal ap = new Panda(pos);
System.out.println((k == ak)  + "␣" + k.equals(ak)  + "␣" + ak.equals(k));
System.out.println((k == p)   + "␣" + k.equals(p)   + "␣" + p.equals(k));
System.out.println((ak == ap) + "␣" + ak.equals(ap) + "␣" + ap.equals(ak));
```

Q 2.7 (1pt) Nous souhaitons introduire une nouvelle fonctionnalité importante : tous les animaux doivent pouvoir effectuer une action qui leur est propre. En pratique, tous les animaux auront une méthode `public void action()`. Comment imposer cette spécification à toutes les classes filles de `Animal` ?

Q 2.8 (1.5pt) Donner le code de la classe `Koala` qui a pour action particulière d'afficher `Je mange des feuilles d'eucalyptus` dans la console. Le Koala a aussi un attribut `poche` de type `Object` qui prend la valeur `null` lorsque la poche est vide (il s'agit aussi de l'état initial du Koala). Donner un accesseur sur cet attribut pour obtenir la valeur.

Q 2.9 (2pts) En considérant qu'il existe aussi une classe `Panda`, donner le code d'un `main` qui :

- crée 20 animaux pouvant être des Koala (probabilité 1/3) ou des Panda (probabilité 2/3)
- parcourt la liste des animaux en exploitant l'accesseur développé en Q2.4 et pour chaque animal :
  - affiche son `id`,
  - effectue son action,
  - affiche le contenu de la poche (pour les `Koala` seulement).

Note : faire très attention aux types des variables, aux tests et aux éventuelles conversions.

Exercice 3 (8.5pts) – Exceptions

On considère un programme qui indique si des livraisons ont pu être effectuées (ou non) dans un magasin en fonction de l'heure à laquelle elles ont été effectuées. On considère les 3 classes correctes suivantes :

```
public class HoraireException extends Exception {
    public HoraireException(int heure, String msg) {
        super(heure+"␣heure"+msg);
    }
}
public class TropTotException extends HoraireException {
    public TropTotException(int heure) {
        super(heure,",␣c'est␣trop␣tot␣!");
    }
}
public class Magasin {
    private int debut, fin;
    public Magasin(int debut, int fin) { this.debut=debut; this.fin=fin; }
    public void livraison(int heure) throws TropTotException, HoraireException {
        if (heure<0 || heure >=24)
            throw new HoraireException(heure,"␣mauvais␣horaire");
        if (heure<debut)
            throw new TropTotException(heure);
        if (heure>=fin)
            throw new HoraireException(heure,",␣c'est␣trop␣tard␣!");
        System.out.println("Article␣livré␣à␣"+heure+"␣heure");
    } }
```

Q 3.1 Soit la classe suivante :

```
23 public class TestHoraire {
24     public static void main(String [] args) {
25         int heure=-1;
26         Magasin mag=new Magasin(9,19);
27         for(int i=0;i<args.length;i++) {
28             try {
29                 heure=Integer.parseInt(args[i]);
30                 mag.livraison(heure);
31             } catch(HoraireException h) {
32                 System.out.println("Désolé␣"+h.getMessage());
33             }
34         }
35         System.out.println("Fin");
36     }
37 }
```

Rappel : `Integer.parseInt(chaine)` lève l'exception `NumberFormatException` (qui est de type `RuntimeException`) quand `chaine` n'est pas un entier.

Q 3.1.1 (2pts) Le code ci-dessus est syntaxiquement correct et compile. Quel est l'affichage obtenu pour :
(a) `java TestHoraire 10 15` (aide : cela affiche 3 lignes)
(b) `java TestHoraire 14 27`
(c) `java TestHoraire 11 abc 16`

Q 3.1.2 (1pt) Est-il intéressant d'ajouter : `catch(TropTotException t) { }` **après** le premier `catch` (entre les lignes 33 et 34) ? Expliquez brièvement, mais précisément votre réponse.

Q 3.2 On modifie la classe `TestHoraire` ainsi :

```
38 public class TestHoraireV2 {
39     public static void main(String [] args) {
40         int heure=-1;
41         Magasin mag=new Magasin(9,19);
42         for(int i=0;i<args.length;i++) {
43             try {
44                 heure=Integer.parseInt(args[i]);
45                 mag.livraison(heure);
46             } catch(TropTotException t) {
47                 System.out.println(t.getMessage()+"␣Revenez␣dans␣une␣heure.");
48                 // On retente la même livraison une heure plus tard
49                 args[i]=(heure+1)+""; // ajout d'une heure à args[i]
50                 i=i-1; // au prochain passage dans la boucle, on aura la même valeur de i
51             } catch(HoraireException h) {
52                 System.out.println("Désolé␣"+h.getMessage());
53             } catch(Exception e) {
54                 System.out.println("Désolé,␣je␣ne␣comprends␣pas␣"+args[i]);
55             }
56         }
57         System.out.println("Fin");
58     }
59 }
```

Quel est l'affichage obtenu dans les cas suivants ?

Q 3.2.1 (1pt) `java TestHoraireV2 11 abc 16` (aide : cela affiche 4 lignes)

Q 3.2.2 (1.5pt) `java TestHoraireV2 7 20` (aide : cela affiche 5 lignes)

Q 3.3 On suppose maintenant que les magasins n'acceptent pas de livraison durant la pause de midi, c'est-à-dire à 12h et à 13h. De plus, on ne veut pas modifier le main de `TestHoraireV2`.

Q 3.3.1 (2pts) Écrire une classe `MidiException` qui :

— permette, **sans modifier le main de** `TestHoraireV2`, de retenter la livraison une heure plus tard

— génère un message qui dépendra de l'heure et sera par exemple :
`"13 heure, c'est la pause !"`.

Note : la solution n'est pas très élégante d'un point de vue architecture logicielle mais vous devez respecter la spécification ci-dessus.

Q 3.3.2 (0.5pt) Donner les instructions qu'il faut ajouter dans la méthode `livraison` de la classe `Magasin`.

Q 3.3.3 (0.5pt) Quel est maintenant l'affichage obtenu par : `java TestHoraireV2 12` ? (aide : cela affiche 4 lignes)

Exercice 4 (21pts) – **Mille bornes**

L'objet de cet exercice est d'implémenter une version allégée du jeu "1000 bornes" pour `n` joueurs.
On considèrera les règles du jeu suivantes :

- R1 : Les cartes sont :
  - soit des kilomètres (cartes "50 kms", "100 kms", "200 kms"),
  - soit des contraintes (cartes "Accident", "Panne d'essence", "Crevaison"),
  - soit des résolutions (cartes "Réparations", "Essence", "Roue de secours").
- R2 : Le plateau de jeu dispose d'un tas de cartes (la "pioche").
- R3 : Chaque joueur possède 6 cartes au début du jeu. Il aura aussi une pile de cartes devant lui durant le jeu où seront posées les contraintes/résolutions. Chaque joueur est associé à un identifiant et un score (décompte des km parcourus jusqu'ici).
- R4 : Chaque kilomètre posé par un joueur lui permet d'augmenter son score.
- R5 : Un joueur peut empêcher un autre joueur de cumuler des kilomètres en posant sur sa pile des contraintes. Le joueur attaqué peut résoudre ces contraintes grâce aux cartes résolutions.
- R6 : Les joueurs jouent chacun leur tour. Ils auront la possibilité d'effectuer une (et une seule) action : poser des kilomètres sur la pile du jeu (et donc d'augmenter leur score) OU poser des contraintes sur la pile d'un autre joueur OU poser des résolutions sur sa propre pile OU piocher une carte. (Remarque : à l'inverse du vrai jeu, les joueurs pourront avoir plus de 6 cartes en main).
- R7 : La partie est terminée quand un joueur arrive à 1000 kilomètres.

Q 4.1 Les cartes. Nous proposons de gérer les différentes cartes du jeu à l'aide de l'héritage.

Q 4.1.1 (1pt) Proposer une hiérarchie de classes pour les cartes. Selon vous, la classe mère doit-elle être abstraite ? Pourquoi (en une phrase) ?

Q 4.1.2 (2pts) Les cartes de kilomètres sont composées d'un attribut entier, les cartes contraintes/résolution d'un message et d'un identifiant qui devra être le même pour la contrainte et la résolution associée (par exemple : accident : id = 1 et réparation : id = 1). Donner le code de la classe mère `Carte` qui ne possède qu'une méthode `toString` rendant la `String "Carte :"`. Donner ensuite le code de la classe `CarteKM` qui :

- possède un constructeur prenant en argument le nombre de KM mais interdit la création de cartes depuis l'extérieur de la classe.
- possède trois méthodes factory nommées `Carte makeCarteXXXKm()` permettant de construire les cartes de kilométrage prévues dans la règle. Bien réfléchir à la signature complète de ces méthodes.
- possède un accesseur sur les kilomètres.
- possède une méthode `toString` exploitant obligatoirement la méthode `toString` de la classe mère et ajoutant une information sur le nombre de kilomètres.

Q 4.1.3 (1pt) Quel est l'intérêt de l'architecture imposée pour la classe précédente ? Donner la ligne d'un `main` permettant de récupérer une carte 100km.

Le code des cartes contrainte et résolution **n'est pas demandé**. Vous noterez que ces classes disposent d'une méthode `public int getId()` permettant de vérifier l'identifiant et donc la compatibilité entre les cartes.

Q 4.2 (4pts) Les joueurs. Donner le code de la classe `Joueur` qui a pour attributs :

- une `main` correspondant aux cartes qu'il peut jouer (une `ArrayList<Carte>`),
- une `Carte` contrainte, initialisée à `null` qui permettra aux autres joueurs d'ajouter des contraintes [1],
- un identifiant géré avec un compteur static,
- une `Strategy str` (qui sera définie dans les questions suivantes),
- un score (entier, exprimé en nombre de km parcouru).

Le joueur possèdera les méthodes suivantes :

- Un constructeur prenant en argument la stratégie et initialisant tous les attributs.
- Un accesseur et un setter sur la carte contrainte.

1. Un joueur ne peut donc être soumis qu'à une seule contrainte. Si un joueur ajoute une contrainte alors qu'il y en a déjà une, la nouvelle remplace l'ancienne.

— Un accesseur sur le score.

— Une méthode `void action(Jeu j)` qui reste vide dans un premier temps.

— Une méthode `toString` décrivant l'identifiant du joueur, son score et l'ensemble de ses cartes ainsi que la contrainte éventuelle du joueur.

— Une méthode `void ajouterCarte(Carte)` qui permet d'ajouter une carte dans la main du joueur.

— Un accesseur sur la main du joueur.

— Une méthode `void jouerCarteKM(Carte)` qui incrémente le score du joueur du nombre de KM de la carte (incrément = 0 si la carte n'est pas une carte KM) et élimine la carte de la main [2].

Donner un premier code de la classe `Joueur`.

Q 4.3 (4pts) Le jeu. Le plateau de jeu est constitué d'un ensemble de joueurs (`ArrayList<Joueur>`) et d'une pioche (`ArrayList<Carte>`). Donner le code de la classe respectant les spécifications suivantes.

— Le jeu est initialisé en donnant seulement une liste de joueurs. On imagine que l'on dispose de la méthode `public static ArrayList<Carte> makePioche()` dans la classe `ToolsJeu` qui permet de générer le paquet de cartes d'origine. N'oubliez pas de distribuer 6 cartes à chaque joueur.

— Il possède une méthode `piocher()` qui retourne la dernière carte de la pioche (et l'enlève de la pioche) ou `null` si la pioche est vide.
Note : vous remarquerez que la méthode `remove(int i)` de `ArrayList` retourne l'élément éliminé.

— Le jeu possède une méthode `Joueur jouer()` qui donne successivement la main à tous les joueurs en invoquant leur méthode `action` tant que le jeu n'est pas fini (ou que le nombre de tour est inférieur à 25). La méthode retourne le vainqueur (ou `null` dans le cas où la limite du nombre de tours est atteinte).
Note : on gagne en élégance en codant la détection de fin de partie dans une méthode à part.

Q 4.4 Stratégies des joueurs. On souhaite développer une panoplie de stratégies pour nos joueurs (i.e. différentes classes). Toutes les stratégies posséderont une méthode `void jouerUnTour(Jeu plateau, Joueur j)`. Cette méthode permettra de jouer un tour de la partie pour le joueur `j`, c'est à dire de poser des cartes contraintes sur les autres joueurs, des cartes résolutions sur sa propre pile, des cartes km pour faire avancer son score ou de piocher.

Q 4.4.1 (1pt) Donner le code associé au fichier `Strategy.java` en choisissant la bonne architecture pour ce concept général.

Q 4.4.2 (1pt) Donner le code de la méthode `void action(Jeu j)` de `Joueur` (maintenant que le fonctionnement de la stratégie est identifié).

Q 4.4.3 (1pt) Donner le code d'une stratégie très simple `StrategyPioche`, qui ne fait que piocher des cartes dans la pioche.

Q 4.4.4 (3pts) Afin de développer des stratégies plus élaborées, nous allons développer deux outils dans une classe `ToolsJeu` :

— Le premier est une méthode `Carte rechercheResolution(ArrayList<Carte> main, int id)` qui recherche si la main possède une carte résolution correspondant à `id`. Si c'est le cas, la carte est retournée, sinon, la méthode retourne `null`.

— Le seconde est une méthode `Carte recherchePlusGdKM(ArrayList<Carte> main)` qui retourne la carte correspondant au plus grand kilométrage dans la main ou `null` si le joueur ne possède aucune carte de kilomètres.

Note : bien réfléchir à la signature des méthodes ainsi qu'aux éventuels problèmes de typage et de cast sur les objets.

Q 4.4.5 (1.5pt) Donner le code d'une stratégie plus avancée qui :

— joue la bonne résolution si le joueur est bloqué par une contrainte et qu'il possède la bonne carte [3].

— sinon, joue ses plus gros KM.

— en cas d'absence de carte KM, pioche une carte dans la pioche du jeu.

Q 4.5 (1.5pt) Donner le code d'un main qui crée 2 joueurs avec des stratégies différentes et lance la partie et affiche le vainqueur.

2. Regarder la documentation de la classe ArrayList pour gagner du temps.
3. Les cartes sont simplement dé-référencées : la contrainte du joueur est mise à null, la résolution est éliminée de la main. Regardez bien la documentation de l'ArrayList pour gagner du temps.

Exercice 5 (18pts) – Modélisation d'un Ascenseur

Dans cet exercice, nous voulons simuler le fonctionnement d'un ascenseur. Dans une première partie, nous allons modéliser la classe `Ascenseur` qui regroupe les fonctionnalités pour les mouvements de l'ascenseur sans se préoccuper de la logique de fonctionnement. Dans la deuxième partie, nous nous intéresserons à cette logique, c'est-à-dire comment les différentes fonctions sont enchaînées pour obtenir un ascenseur opérationnel.

Q 5.1 Un ascenseur dessert `nb_etages` étages qui est un entier entre 0 et un nombre constant dépendant de l'ascenseur. Il se déplace uniquement verticalement à une vitesse précisée par une constante `vitesse` dépendant de l'ascenseur, et sa position est dénotée par un réel `hauteur`. On considère qu'il est arrivé à un étage `e` donné lorsque la hauteur de l'ascenseur est égale à l'étage, plus ou moins une constante réelle `PRECISION=0.05` qui ne dépend pas de l'ascenseur. Pour simuler l'état de la porte de l'ascenseur, une variable `positionPorte` est utilisée entre 0. et 1. correspondant au pourcentage d'ouverture de la porte : à 0 la porte est fermée, à 1 elle est complètement ouverte. Une constante propre à chaque ascenseur `vitesse_porte` précise la vitesse d'ouverture et de fermeture de la porte et une constante `temps_ouverture` précise le temps pendant lequel la porte reste ouverte (la fermeture est automatique).
Les boutons de l'ascenseur seront représentés par un tableau de booléens `boutons`, une case pour chaque étage : lorsque une case est à `true`, l'arrêt à cet étage est demandé. Pour simplifier, ce tableau représente à la fois les boutons dans l'ascenseur et ceux sur les paliers. Enfin, une variable entière `destination` sert à préciser le prochain étage où s'arrête l'ascenseur. Sa valeur est fixée par une stratégie interne de l'ascenseur en fonction des boutons appuyées et en fonction du type d'ascenseur par l'appel de la méthode `void choisirDestination()`. Lorsqu'aucune destination n'est en cours (aucun bouton n'est appuyé), la valeur de la variable est fixée à -1.
Deux types d'ascenseur sont envisagés :

- l'ascenseur *Zig* (classe `AscenseurZig`), dont la méthode `choisirDestination()` fixe la destination à l'étage le plus bas dont le bouton est allumé. Il s'agit d'une stratégie très naïve : quelle que soit sa position, l'ascenseur *Zig* commence par desservir l'étage le plus bas demandé.
- l'ascenseur *Zag* (classe `AscenseurZag`), représente la stratégie symétrique qui fixe la destination à l'étage le plus haut dont le bouton est allumé.

Les méthodes et accesseurs suivants sont donnés dans le code ci-dessous : la méthode `monte()` qui permet de faire monter l'ascenseur pendant un pas de temps, la méthode `descend()` qui permet de faire descendre l'ascenseur pendant un pas de temps, les méthodes `fermePorte()` et `ouvrePorte()` qui permettent d'agir sur la porte pendant un pas de temps. Ces 4 méthodes sont les actions atomiques que peut faire l'ascenseur pendant une unité de temps.

```
public void fermePorte(){positionPorte = Math.max(0,positionPorte-vitesse_porte);}
public void ouvrePorte(){positionPorte = Math.min(1.,positionPorte+vitesse_porte);}
public void monte(){hauteur = Math.min(hauteur+vitesse,nb_etages);}
public void descend(){hauteur = Math.max(hauteur-vitesse,0);}
public double getHauteur(){return hauteur;}
public int getDestination(){return destination;}
public double getHauteur(){return hauteur;}
```

Q 5.1.1 (2.5pts) Donner le début de la classe `Ascenseur` : les déclarations des variables (les constantes seront publiques) et le constructeur à 4 paramètres : le nombre d'étages, la vitesse de l'ascenseur, la vitesse d'ouverture de la porte, le temps d'ouverture de la porte.

Q 5.1.2 (2 pts) Donner le code pour les méthodes de la classe suivantes :

- `appuyerBouton(int c)` qui permet de simuler l'appuie sur le bouton de l'étage `c`.
- la méthode `bouge()` qui permet de faire monter ou descendre l'ascenseur pendant un pas de temps en fonction de la position de l'ascenseur et de la destination.
- les méthodes `boolean porteFermee()` et `boolean porteOuverte()` qui testent si la porte est complètement fermée et respectivement ouverte.
- la méthode `boolean estAEtage(int i)` qui teste si l'ascenseur est à l'étage `i`.
- la méthode `void choisirDestination()` telle que décrite ci-dessus.

Q 5.1.3 (1.5 pts) Donner les classes `AscenseurZig` et `AscenseurZag`.

Q 5.2 (7 pts) Après les questions précédentes, l'ascenseur dispose de toutes les commandes pour le faire fonctionner mais il manque le processus de contrôle. À chaque pas de temps, une boucle externe va appeler une nouvelle méthode de l'ascenseur : `void update()`. A chaque pas de temps, une action (et une seule) est autorisée parmi les actions atomiques : monter, descendre, ouvrir ou fermer la porte d'un pas.
Toute la partie logique pourrait être codée dans cette méthode `update`, mais cela aboutirait à une méthode peu flexible et peu lisible. On préfère utiliser un *design pattern* appelé *Machine à états.*
Un ascenseur peut être dans différents états :

- `Attente` : les portes de l'ascenseur sont fermées, l'ascenseur reste dans cet état tant qu'une destination n'est pas disponible. Lorsque la prochaine destination est connue, il passe dans l'état `EnMouvement`.
- `EnMouvement` : l'ascenseur monte ou descend selon l'étage de destination. Lorsqu'il arrive à destination, il passe dans l'état `OuverturePorte`.
- `OuverturePorte` : l'ascenseur ouvre sa porte ; une fois ouverte, il passe dans l'état `PorteOuverte`.
- `PorteOuverte` : l'ascenseur maintient la porte ouverte pendant `TEMPSOUVERTURE` unités de temps ; lorsque le temps est écoulé, il passe dans l'état `FermeturePorte`.
- `FermeturePorte` : l'ascenseur ferme la porte ; lorsque la porte est fermée il passe dans l'état `Attente`.

On utilise une interface `Etat` pour spécifier les fonctionnalités des états. Chaque état comporte une méthode `void updateEtat(Ascenseur a)` qui permet de tester s'il faut changer d'état et d'exécuter une action atomique de l'ascenseur. L'ascenseur comporte une variable `Etat etat` pour stocker l'état courant, un setteur `void setEtat(Etat etat)` et la méthode `void update()` qui appelle entre autre la méthode `updateEtat` de l'état courant. L'ascenseur n'a jamais à fixer l'état courant (sauf lors de sa création), ce sont les états qui fixent l'état suivant de l'ascenseur.
Donner le code des classes qui représentent les différents états.

Q 5.3 (3pts) A-t-on besoin de créer une nouvelle instance de chaque état à chaque fois ? Quelle solution proposez-vous ? Donner le code pour la classe `Attente`.

Q 5.4 (2pts) On veut être certain que lorsque l'ascenseur est en mouvement la porte est bien fermée. Si ce n'est pas le cas, une exception `AscenseurException` doit être levée avec le message *l'ascenseur est bloqué à la hauteur h.* Le contrôle doit s'effectuer dans la méthode `bouge`. L'ascenseur doit se mettre alors dans un nouvel état : `EnPanne`, état terminal qui ne fait rien. Donner la classe `AscenseurException` et les modifications à apporter.

Documentation `ArrayList<E>` (extrait)

- constructeur sans paramètre
- `boolean add(E e)` ajoute `e` à la fin de liste
- `void add( ajouté quelques remarquesint index, E element)` ajoute `element` à la position `index` et décale les élements suivants ; lève l'exception `IndexOutOfBoundsException` si $index < 0 \;||\; index > size()$
- `E get(int index)` retourne l'élément à la position `index` ; lève l'exception `IndexOutOfBoundsException` si $index < 0 \;||\; index >= size()$
- `E remove(int index)` supprime de la liste l'élément à la position `index` et le retourne
- `boolean remove(Object o)` supprime la première occurrence dans la liste de l'objet référencé par `o` si l'objet est présent dans la liste et retourne `true` ; si l'objet n'est pas dans liste, retourne `false`
- `int size()` retourne le nombre d'éléments de la liste

Il faut ajouter `import java.util.ArrayList;` au début du fichier.