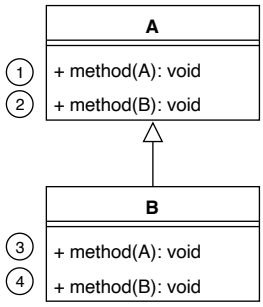
LU2IN002 – 2019-2020 – Examen Session 1

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 ?

Remarque : bug dans le texte distribué aux étudiants, lire "new" en ligne 11 et non pas "naw"...
Barème : 0.5pt par par valeur correcte.
Au total
Dans la boucle for : 1 A + 3 B
+ 1 A après = 2 A et 3 B
A la fin : 2 B ont été dé-référencés, il ne reste donc que 2 A et 1 B.

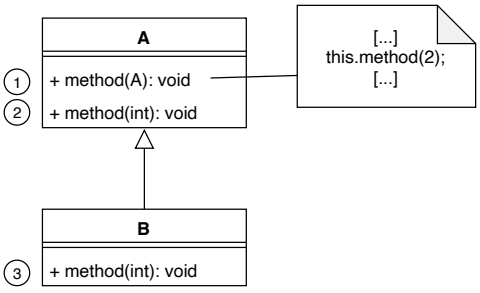
Q 1.2 (1.5pt) Dans le `main` de droite à la **ligne 4**. Quelles sont les méthodes pré-sélectionnées par le **compilateur** ? Quelles sont les méthodes exécutées par la **JVM** ? Donner simplement 2 séries de 3 numéros par rapport à la numérotation proposée sur le diagramme UML.

Barème : 0.25pt par valeur correcte.
Présélection : 1 2 1
Exécution : 3 4 3

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.

Barème : 0.25pt par valeur correcte.
Présélection : 1 1 1
Exécution : 3 3 3

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 ?



Barème : 0.5pt par valeur.
Exécution : 2 3 3

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` ?

Barème : 0.25pt si "oui" + 0.25pt si justification.
Oui, les attributs correspondant sont final : ils sont sécurisés

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 ?

Barème : tout (si justifié) ou rien.
Non, final ne protège que de l'affectation, pas de l'invocation de méthode sur l'objet. Pas de problème, pas besoin de l'enlever

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.

Barème : 0.75pt pour la structure correcte + 0.25pt pour le calcul correct des nouvelles coordonnées.
On est obligé de retourner un nouveau vecteur car les coordonnées des opérandes sont constantes.

```
1 public Vecteur add(Vecteur v){
2     return new Vecteur(x + v.x, y+v.y);
3 }
```

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.

Remarque : dit aux étudiants durant l'examen : lire `id` au lieu de `ind`.

Barème : 0.5pt explication + 1pt static + 0.5pt implem
Stocker toutes les instances créées durant l'exécution du programme

```
1 public static Animal getAnimal(int ind){
2   Animal a = null;
3   if(ind<all.size() && ind>=0)
```

```
4     a = all.get(ind);
5   return a;
6 }
```

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.

Barème :

- 1pt pour equals Vecteur + 1pt pour penser à la composition dans equals de Animal + 1pt pour clone Animal
- -1 si clone/teste sur id et all.
- Elimination de tous les points en cas de mauvaise signature sur equals (Vecteur au lieu de Object...)

Note : le clone sur le Vecteur est facultatif. Avec des attributs constants il ne peut rien arriver de mal même si on ne le fait pas.

```
1 // dans Vecteur
2 public boolean equals(Object o){
3   if(o==null) return false;
4   if(o.getClass() != this.getClass()) return false;
5   Vecteur v = (Vecteur) o;
6   return v.x == x && v.y == y;
7 }
8 public Vecteur clone(){
9 return new Vecteur(x, y);
10 }
11
12 // Dans Animal
13 public boolean equals(Object o){
14   if(o==null) return false;
15   if(o.getClass() != this.getClass()) return false;
16   Animal a = (Animal) o;
17   return a.position.equals(position);
18 }
19 public Animal clone(){
20 return new Animal(position.clone());
21 }
```

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 :

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

Barème : -0.5 par faute
false true true
false false false
false false false

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` ?

Barème : tout pour abstract (1pt) ou rien (0pt)
Il faut ajouter une méthode abstract dans Animal. Cela rend la classe abstract

```
public abstract void action();
```

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.

Barème : 0.5pt (entête et constructeur globalement)+ 0.25pt (pour appel super()) + 0.5pt (poche et getPoche() : bon type Object) et 0.25pt pour action()

```
public class Koala extends Animal{
  private Object poche;
  public Koala(Vecteur pos){
    super(pos);
    poche = null; // facultatif
  }
  public Object getPoche(){return poche;}
  public void action(){
    System.out.println("Je mange des Eucalyptus");
  }
}
```

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 Q**??** 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.

Barème : 1pt pour la boucle de création correcte + 1pt pour la boucle de parcours (dont 0.5pt pour un instanceof correctement utilisé).
On peut accepter getClass() au lieu de instanceof.

```
public class Test{
  public static void main(String[] args){
    Animal[] tab = new Animal[20];
    for(int i=0; i<tab.length; i++)
      if(Math.random()<0.33)
        tab[i] = new Koala(new Vecteur());
      else
        tab[i] = new Panda(new Vecteur());

    for(Animal a:tab){
      System.out.println(a.id);
      a.action();
      if(a instanceof Koala)
        System.out.println(((Koala) a).getPoche());
    }

  }
}
```

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 :

```
1 public class HoraireException extends Exception {
2     public HoraireException(int heure, String msg) {
3         super(heure+" heure"+msg);
4     }
5 }
6 public class TropTotException extends HoraireException {
7     public TropTotException(int heure) {
8         super(heure,", c'est trop tot !");
9     }
10 }
11 public class Magasin {
12     private int debut, fin;
13     public Magasin(int debut, int fin) { this.debut=debut; this.fin=fin; }
14     public void livraison(int heure) throws TropTotException, HoraireException {
15         if (heure<0 || heure >=24)
16             throw new HoraireException(heure," mauvais horaire");
17         if (heure<debut)
18             throw new TropTotException(heure);
19         if (heure>=fin)
20             throw new HoraireException(heure,", c'est trop tard !");
21         System.out.println("Article livré à "+heure+" heure");
22     } }
```

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)

Barème : 0.5pt si deux articles livrés + "Fin" (-0.25pt par ligne oubliée)

```
Article livré à 10 heure
Article livré à 15 heure
Fin
```

(b) `java TestHoraire 14 27`

Barème : 0.5pt si un article livré (0.25pt) + "mauvais horaire" (0.25pt)

```
Article livré à 14 heure
Désolé 27 heure mauvais horaire
Fin
```

(c) `java TestHoraire 11 abc 16`

Barème : 0.5pt si un article livré + 0.5pt pour l'exception (-0.5pt si affichage pour "16")

```
Article livré à 11 heure
Exception in thread "main" java.lang.NumberFormatException: For input string: "abc"
```

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.

Barème : 0.5pt bonne réponse + 0.5pt si l'explication est correcte

Non, car comme `TropTotException` hérite de `HoraireException`, `catch(HoraireException h)` capture les exceptions de type `HoraireException` et aussi de type `TropTotException`, en conséquence, le 2ième `catch` ne serait jamais appelé.

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)

Barème : 1pt (-0.5pt par ligne fausse ou oubli de ligne)

```
Article livré à 11 heure
Désolé, je ne comprends pas abc
Article livré à 16 heure
Fin
```

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

Barème : 1.5pt (-0.5pt par ligne fausse ou oubli de ligne)

```
7 heure, c'est trop tot ! Revenez dans une heure.
8 heure, c'est trop tot ! Revenez dans une heure.
Article livré à 9 heure
Désolé 20 heure, c'est trop tard !
Fin
```

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.

Barème : 0.5pt pour `extends TropTotException` + 1pt constructeur (avec super) + 0.5pt pour getMessage()

Il faut hériter de `TropTotException`.

```
1 public class MidiException extends TropTotException {
2     private int heure;
3     public MidiException (int heure) {
4         super(heure);
5         this.heure=heure;
6     }
7     public String getMessage() {
8         return heure+" heure, c'est la pause !";
9     }
10 }
```

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

Barème : 0.5pt (0.25pt pour le if, 0.25pt pour le throw)

On ajoute après la ligne 20 :

```
1 if (heure == 12 || heure == 13)
2         throw new MidiException(heure);
```

Remarque : inutile de rajouter `throws MidiException`, car `MidiException` hérite de `TropTotException`, mais on peut le faire.

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

Barème : 0.5pt pour la totalité (ou 0 sinon). Compter correct si c'est cohérent avec ce qui a été fait avant

```
12 heure, c'est la pause midi ! Revenez dans une heure.
13 heure, c'est la pause midi ! Revenez dans une heure.
Article livré à 14 heure
Fin
```

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) ?

Barème : 0.5pt pour la hiérarchie + 0.5pt pour les explications sur la classe mère abstraite

— Carte <<ABS>>
  — CarteContrainte
  — CarteKM
  — CarteResolution

On peut faire plus de niveau, mais ce n'est pas utile.
La classe mère doit être abstraite car c'est un concept qui n'a pas vocation à être implémenté (même s'il n'y a pas de méthode abstraite dans le cas présent)

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.

Barème :

— 0.25pt pour la classe Carte

— 0.5pt pour le constructeur et la gestion de nbKM dans CarteKM

— 0.25pt pour toString() de CarteKM (faux si pas de super())

— 1pt pour les 3 méthodes makeCarteXXX() (retirer 0.5pt si pas static).

```
1 public abstract class Carte {
2         public String toString() {
3                 return "Carte␣:";
4         }
5 }
6
7 public class CarteKM extends Carte {
8
9         private int nbKm;
10
11        private CarteKM(int nbKM){
12                this.nbKm=nbKM;
13        }
14        public int getKm(){
15                return nbKm;
16        }
17        public String toString(){
18                return super.toString() + nbKm + "km";
19        }
```

```
20          public static Carte makeCarte50Km(){
21              return new CarteKM(50);
22          }
23          public static Carte makeCarte100Km(){
24              return new CarteKM(100);
25          }
26          public static Carte makeCarte200Km(){
27              return new CarteKM(200);
28          }
29
30  }
```

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.

Barème : 0.5pt pour l'explication + 0.5pt pour la syntaxe

Le but de cette architecture est d'interdire à l'utilisateur la création de cartes qui n'existent pas. Il s'agit d'une architecture sécurisée.

```
1 Carte c_km = CarteKM.makeCarte100Km();
```

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.
- 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`.

Barème : 1pt cpt static + 1pt constructeur / accesseurs + 1pt (toString+ajoutercarte) + 1pt incrément du score

```
1 public class Joueur {
2         public static int cpt = 0;
3         private int idJoueur;
4         private ArrayList<Carte> main;
```

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.
2. Regarder la documentation de la classe ArrayList pour gagner du temps.

```
5         private Carte contrainte;
6         private int score;
7         private Strategy str;
8
9         public Joueur(Strategy str){
10                this.str = str;
11                idJoueur=cpt;
12                cpt++;
13                main = new ArrayList<Carte>();
14                contrainte = null;
15                score = 0;
16        }
17
18        public void action(Jeu j) {
19                //vide pour l'instant
20        }
21
22        public int getScore(){
23                return score;
24        }
25
26        public ArrayList<Carte> getMain(){return main;}
27
28    public String toString(){
29        String s = "";
30        s += idJoueur + "␣:␣" +score +"\nCartes␣:␣";
31        for(Carte c:main)
32                s+=c;
33        return s;
34    }
35
36        public void ajouterCarte(Carte c) {
37                main.add(c);
38        }
39
40        public Carte getContrainte() {
41                return contrainte;
42        }
43
44        public void setContrainte(Carte contrainte) {
45                this.contrainte = contrainte;
46        }
47
48        public void jouerCarteKM(Carte c) {
49                if(c instanceof CarteKM)
50                        score += ((CarteKM) c).getKm();
51                main.remove(c);
52        }
53 }
```

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.

Barème : 1pt constructeur + 1pt pour la détection de fin de partie + 1 pt pour le reste de jouer + 0.5pt pour piocher + 0.5pt pour l'élégance générale du code (constantes, style du while...)

```java
public class Jeu {
        private ArrayList<Joueur> joueurs;
        private ArrayList<Carte> pioche;
        public final static int NBCARTES = 6;
        public final static int NBMAXTOURS = 25;
        public final static int NBKM = 1000;

        public Jeu(ArrayList<Joueur> joueurs){
                this.joueurs = joueurs;

                pioche = ToolsJeu.makePioche();

                for(Joueur j:joueurs)
                        for(int i =0; i<NBCARTES; i++)
                                j.addCartes(piocher());

        }

        public Joueur jouer(){
                int tours = 0;
                while (tours < NBMAXTOURS && findepartie() == null){
                        System.out.println("tour " + tours);
                        tours++;
                        for (Joueur j : joueurs){
                                j.action(this);
                        }
                }
                return findepartie();
        }
        public Carte piocher(){
                if(pioche.size()>0)
                        return pioche.remove(0);
                return null;
        }
        private Joueur findepartie() {
                for(Joueur j: joueurs) {
                        if(j.getScore()>=NBKM)
                                return j;
                }
                return null;
        }
}
```

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.

Barème : 0.75pt si interface ou classe abstraite, + 0.25pt pour la bonne méthode. Il faut une interface (mais on ne pénalise pas si classe abstraite pure)

```java
public interface Strategy {
        public void jouerUnTour(Jeu plateau, Joueur j);
}
```

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).

Barème : 0.75pt pour l'appel correct + 0.25pt pour l'ensemble

```java
// Dans Joueur
```

```
2 public void action(Jeu j) {
3                 str.jouerUnTour(j, this);
4         }
```

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

Barème : 0.25pt pour la signature (implements ou extends selon la réponse à la Q4.4.1) + 0.5pt pour la méthode + 0.25pt pour la gestion de c==null.

```
1 public class StrategyPioche implements Strategy{
2         public void jouerUnTour(Jeu plateau, Joueur j) {
3                 Carte c = plateau.piocher();
4                 if(c != null)
5                         j.ajouterCarte(c);
6         }
7 }
```

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.

Barème : 1pt par méthode + 0.5pt dans chaque méthode pour instanceof/getClass()

```
1         public static Carte rechercheResolution(ArrayList<Carte> main, int id) {
2                 for(Carte c: main)
3                         if(c instanceof CarteResolution)
4                                 if(((CarteResolution) c).getId() == id)
5                                         return c;
6                 return null;
7         }
8         public static Carte recherchePlusGdKM(ArrayList<Carte> main) {
9                 int max = -1;
10                Carte retour = null;
11                for(Carte c: main)
12                        if(c instanceof CarteKM)
13                                if(((CarteKM) c).getKm()>max) {
14                                        retour = c;
15                                        max = ((CarteKM) c).getKm();
16                                }
17                return retour;
18        }
```

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.

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.

Barème : 1pt si la solution proposée est correcte "globalement" (ie. l'idée est là) + 0.5pt si tout est parfait.

```
1 public class StrategyAvancee implements Strategy{
2         public void jouerUnTour(Jeu plateau, Joueur j) {
3                 if(j.getContrainte() != null) {
4                         Carte resol = ToolsJeu.rechercheResolution(j.getMain(), ((
                              CarteContrainte) j.getContrainte()).getId());
5                         if(resol != null) {
6                                 j.getMain().remove(resol);
7                                 j.setContrainte(null);
8                         }
9                 }
10                else {
11                        Carte km = ToolsJeu.recherchePlusGdKM(j.getMain());
12                        if(km != null)
13                                j.jouerCarteKM(km);
14                        else
15                                j.ajouterCarte(plateau.piocher());
16                }
17
18        }
19
20 }
```

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.

Barème : 0.5pt si l'instanciation des joueurs est correcte (2 stratégies) + 0.5pt le lance de la partie + 0.5pt l'affichage du vainqueur.
Adapter en fonction des réponses fournies précédemment : donner les points si la solution est correcte au regard des solutions données aux questions précédentes.

```
1 public class testJeu {
2
3         public static void main (String[] args){
4
5                 Strategy str1 = new StrategyPioche();
6                 Strategy str2 = new StrategyAvancee();
7
8                 ArrayList<Joueur> all = new ArrayList<Joueur>();
9                 all.add(new Joueur(str1));
10                all.add(new Joueur(str2));
11
12                Jeu jeu = new Jeu(all);
13
14                jeu.jouer();
15        }
16 }
```

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.

Barème : 1pt abstract + 1pt pour les static & final + 0.5pt pour le reste

```
1 public  abstract class Ascenceur {
2     protected double hauteur = 0;
3     protected int destination = -1;
4     protected boolean [] boutons;
5     public  final int nb_etages;
6     public final double vitesse;
7     public static final double PRECISION=0.05;
8     public final int temps_ouverture;
9     private double positionPorte = 0.;
10    public final double vitesse_porte;
11
12    public  Ascenceur(int etageMax, double vitesse, double ouverture, double temps ){
13        nb_etages=etageMax;
14        boutons = new boolean [etageMax+1];
15        this.vitesse = vitesse;
16        temps_ouverture = temps;
17        vitesse_porte = ouverture;
18    }
```

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.

Barème : -0.5 pts par item faux

```
1 public void appuyerBouton(int c){
2         boutons[c] = true;
3     }
4 public void bouge(){
5    if ((getHauteur()-getDestination())>0) descend();
6         else monte();
7 }
8 public boolean porteFermee(){
9         return (positionPorte ==0.);
10 }
11 public boolean porteOuverte(){
12         return (positionPorte ==1.);
13 }
14 public boolean estAEtage(int i){
15     return (Math.abs(getHauteur()-i)<PRECISION);
16 }
17 public abstract choisirDestination();
```

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

Barème : 0.5pt pour le constructeur + 0.5pt si pas de problème pour l'héritage/abstract + 0.5pt pour le corps des fonctions.

```
1 public class AscenseurZig extends Ascenseur {
2
3     public  AscenseurZig(int etageMax, double vitesse, double ouverture, double temps ){
4         super(etageMax,vitesse,ouverture,temps);
5     }
6     public void choisirDestination(){
7             destination = -1;
8             for (int i=0;i<=nb_etages;i++)
9              if (boutons[i]){
10                 destination = i;
11                 return;
12             }
13     }
14 }
15
16 public class AscenseurZag extends Ascenseur {
17     public  AscenseurZag(int etageMax, double vitesse, double ouverture, double temps ){
18         super(etageMax,vitesse,ouverture,temps);
19     }
20    public void choisirDestination(){
21              destination = -1;
22             for (int i=nb_etages;i>=0;i--){}
23              if (boutons[i]){
24                  destination = i;
25                  return;
26              }
27     }
28 }
```

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.

Barème : 1pt par classe + 2pt pour PorteOuverte.

```
1 public interface Etat {
2     public void updateEtat();
3 }
4
5 public class OuvrePorte extends Etat {
6     public void updateEtat(Ascenceur a){
7         if(a.porteOuverte())
8             a.setEtat(new PorteOuverte());
9         else a.ouvrePorte();
10    }
11 }
12
13 public class FermePorte extends Etat {
14    public void updateEtat(Ascenceur a){
15        if (a.porteFermee())
16            a.setEtat(new Attente());
17        else a.fermePorte();
18    }
19 }
20
21 public class EnMouvement implements Etat {
22    public void updateEtat(Ascenceur a){
23        if (a.estAEtage(a.getDestination())){
24            a.setEtat(new OuvrePorte());
25        }
26        a.bouge();
27    }
28 }
29
30 public class Attente implements Etat {
31    public void updateEtat(Ascenceur a){
32        if (a.getDestination()<0)
33            a.choisirDestination();
34        else a.setEtat(new EnMarche());
35    }
36 }
37
38 public class PorteOuverte implements Etat {
39    private int cpt = 0;
40    // ce serait plus propre avec un constructeur
41
42    public void updateEtat(Ascenceur a){
43        if (cpt>=a.temps_ouverture){
44            a.setEtat(new FermePorte());
45        }
46        cpt++;
47    }
48 }
```

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`.

Barème : 1pt pour singleton + 0.5pt pour PorteOuverte + 1pt pour le code.
réponse : Non sauf pour PorteOuverte, on peut faire des singletons.

```
1 public class Attente implements Etat {
2     private Attente(){}
3     public static Attente EtatAttente = new Attente();
4 }
```

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.

Barème : 0.5 pour chaque méthode/classe.

```
1 public void bouge() throws  AscenseurException {
2     if (!porteFermee()) throw new AscenseurException("Ascenseur est bloqué à la hauteur"+
          getHauteur());
3     ...
4 }
5
6 // Dans EnMarche
7
8 public void updateEtat(Ascenseur a){
9     try{
10    ...
11    }catch(AscenseurException e){
12        a.setEtat(new EnPanne());
13    }
14 }
15
16 public class EnPanne implements Etat(){
17    public void updateEtat(Ascenseur a){}
18 }
19
20 public class AscenseurException extends Exception{
21    public AscenseurException(String msg){super(msg);}
22 }
```

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 $index < 0 \mid\mid index > size()$
- `E get(int index)` retourne l'élément à la position `index` ; lève l'exception `IndexOutOfBoundsException` si $index < 0 \mid\mid index >= size()$
- `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.