LU2IN002 – 2021-22 – Examen Deuxième Chance

13 juin 2022 – Durée : 1 heure 30

Seul document autorisé: **feuille A4 manuscrite, recto-verso.**
Pas de calculatrice ou téléphone. Barème indicatif sur 45 (1pt = 2 minutes).
Le sujet est long, les points au delà de 45 sont des bonus

Exercice 1 – Gestion d'un système de tirage de boules de couleur (7 pts)

```java
public class Boule {
    private String couleur;

    public Boule(String couleur) {
        this.couleur = couleur;
    }
}
```

```java
public static void main(String[] args){
    Boule b1 = new Boule("rouge");
    Boule b2 = new Boule("jaune");
    Boule b3 = new Boule("bleue");
    Boule b4 = b1;
}
```

Q 1.1 (0.5 pt) A l'issue de l'exécution du code ci-dessus, combien y a-t-il de variables et d'instances de `Boule` ?

Barème 0.5/0
4 variables, 3 instances

Q 1.2 (1 pt) Donner la ligne de code pour créer un tableau de `Boule` (nommé `urne`) contenant les 4 références précédentes.

```java
Boule[] urne = {b1, b2, b3, b4};
// ou
Boule[] urne = new Boule[4];
urne[0] = ...
```

Q 1.3 (1.5 pt) Donner la ligne de code pour tirer aléatoirement une boule dans l'urne (selon la loi usuelle uniforme). Quelle est la probabilité de tirer une boule rouge ?
Note : utiliser la fonction `Math.random()` qui retourne un réel entre 0 et 1 (non compris).

1 pt pour le code
-1 si erreur de parenthèse
-0.5 si 4 à la place de urne.length

```java
Boule rand = urne[(int)(Math.random()*urne.length)];
```

0.5 pt pour : 1 chance sur 2 de tirer une boule rouge dans cette urne

Q 1.4 (2 pts) Deux erreurs se sont glissées dans la classe suivante : la première empêche la compilation, la seconde provoque un dysfonctionnement (la méthode de génération de boule renvoie toujours des boules *rouges*).
Donner les corrections à effectuer.

```java
public class BouleRandomFactory {
    public final String[] couleurs = {"rouge","jaune","bleue","verte","orange","violette"};

    public static Boule build(){
        return new Boule(couleurs[(int) Math.random()*couleurs.length]);
    }
}
```

1 pt : Attribut static
1 pt : parenthèses après le (int)

Q 1.5 (2 pts) Donner le code de la méthode **standard** `equals` de la classe `Boule`.
Note : l'attribut `couleur` n'est jamais `null` (mais c'est un objet).

Barème dur : j'ai indiqué en cours qu'ils auraient cette question lors de l'examen
Ne pas hésiter à mettre 0 si c'est très approximatif

```
1     public boolean equals(Object obj) { // -3 si mauvaise signature
2         if (this == obj)      return true; // OPT
3         if (obj == null)      return false; // -1 si oubli
4         if (getClass() != obj.getClass())   return false; // -3 si mauvaise gestion du test et/
              ou cast
5         Boule other = (Boule) obj;
6         if (! couleur.equals(other.couleur)) // -1 si == a la place de .equals
7             return false;
8         return true;
9     }
```

Exercice 2 – Nozama (12 pts)

Pour gérer un catalogue de produits, un vendeur en ligne définit le programme suivant :

```
1 public class Livre{
2  private String titre;
3  public Livre(String titre){
4   this.titre = titre;
5  }
6 }
7 public class BD extends Livre{
8  private boolean couleur;
9  public BD(String titre, boolean couleur){
10  super(titre);
11  this.couleur = couleur;
12 }
13 }
```

```
14 public static void main(String[] args){
15   Livre[] tab = new Livre[6];
16   for(int i=0; i<tab.length; i++){
17     if(i%2==0)
18       tab[i] = new Livre("Livre␣"+i);
19     else
20       tab[i] = new BD("BD␣"+i, true);
21   }
22   tab[0] = tab[2];
23   Livre a = new Livre("Livre␣18");
24   tab[3] = a;
25 }
```

Q 2.1 (2pts) Dans le `main`, combien d'instances de `Livre` et de `BD` ont été créées au total ? Combien en reste-t-il à l'issue de l'exécution du programme ?

Barème : 0.5pt par par valeur correcte.
Au total
Sortie de la boucle for : 3 Livre (i pair) + 3 BD (i impair) + 1 livre
= 4L+3 BD
Après Ligne 22 : élimination du livre qui est référencé dans tab[0] (reste 0 Livre et 4 BD)
Après Ligne 23 : 1 Livre créé (donc bilan : reste 1 Livre et 4 BD)
Après Ligne 24 : élimination d'une BD (référencée dans tab[3]) : reste
= 3l + 3BD

Q 2.2 (2.5pts) On envisage d'ajouter une 26ème ligne de code dans le main. Pour chaque proposition (=une des ligne ci-dessous), indiquer (sans justification) : les problèmes de compilation ; les problèmes d'exécution ; les éventuels les affichages.
Toutes les propositions sont indépendantes, on ne considère pas le fait d'ajouter plusieurs lignes.

```
26 tab[0] = null;
27 Livre c = tab[1];
28 BD b = tab[3];
29 System.out.println(tab[2].toString());
30 System.out.println(tab[6]);
```

```
31 Livre[] tab2 = tab; tab2 = null; tab2[3] = tab2[1];
32 Livre[] tab2 = tab; tab2 = null; tab[3]  = tab[1];
33 Livre[] tab2 = tab; tab2[0] = null; tab[0].toString();
34 System.out.println(tab[0].equals(tab[1]));
35 System.out.println(tab[0].equals(tab[2]));
```

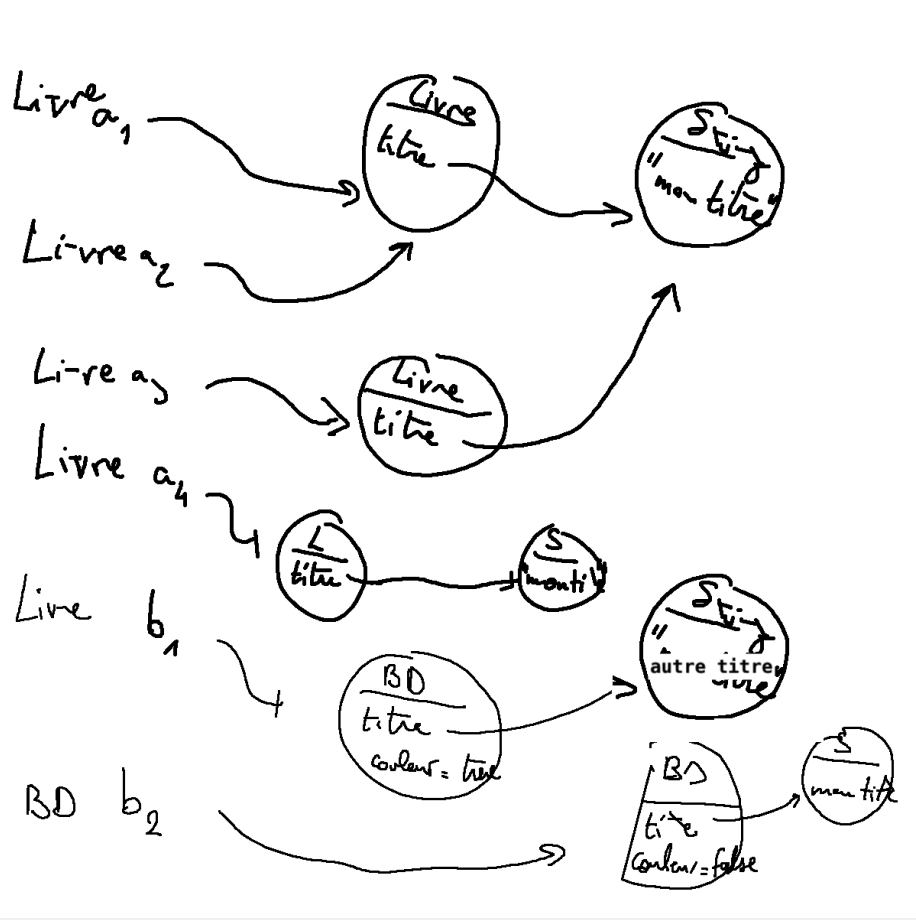
-0.5 par faute = 0 si la moitié est fausse
l29-30 : -0.25 si erreur sur la classe (BD/Livre)

26	OK	31	Exception : tab2 est null pour la dernière instr.
27	OK	32	OK. Pas de souci dans les affectations.
28	Err compil : un Livre n'est pas une BD	33	Exception : tab2[0] est null pour la dernière instr.
29	OK. `BD@qslkdjhfql`	34	OK. Rend false car objets différents en mémoire
30	Err exec OutfBound `BD@plmqslv`	35	OK. Rend true car même référence.

Q 2.3 (2pts) Dessiner le diagramme mémoire des variables et instances à l'issue de l'exécution du code suivant en prenant en compte la nature objet de `String`.

```
1 String s = "mon titre";
2 Livre a1 = new Livre(s);
3 Livre a2 = a1;
4 Livre a3 = new Livre(s);
5 Livre a4 = new Livre("mon titre");
6 s = "un autre titre";
7 boolean couleur = true;
8 Livre b1 = new BD(s, couleur);
9 couleur=false;
10 BD b2 = new BD("mon titre", couleur);
```

0.5 pour les types de variables
0.5 pour les types objets + références Livre/BD
1 pour la gestion des Strings

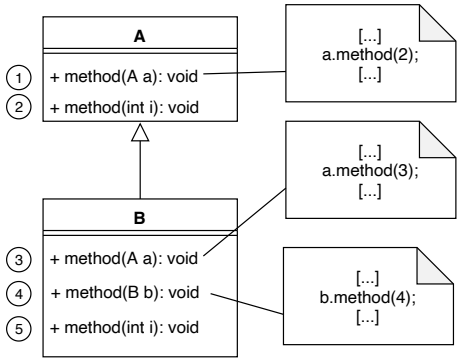


Q 2.4 (2pts) On souhaite que chaque ouvrage (`Livre` ou `BD`) soit associé à un identifiant entier unique immutable, ce qui simplifie les inventaires. Réfléchir à la meilleure approche : faut-il ajouter un/des attribut(s) dans `Livre` seulement ou dans les deux classes ?
Donner la ou les modifications de code qui instancie(nt) cet identifiant.

0.5pt : il faut que ce soit dans Livre et QUE dans Livre
0.5pt pour la ligne du constructeur
1pt pour les attributs correctement déclaré

```
1 public class Livre{
2  private String titre;
3
4  private int id;
5  private static int cpt = 0; // Static + init
6
7  public Livre(String titre){
8   this.titre = titre;
9   id = cpt++;
10  }
11 }
```

Q 2.5 (3.5pts) En considérant l'architecture de classes ci-contre et le `main` suivant, indiquer pour chaque ligne les appels de fonctions qui sont effectués lors de l'exécution.



Exemple de réponse pour la ligne 7 : ①, ②
La ligne 7 correspond à une invocation de la fonction ① qui, elle-même invoque la fonction ②

Attention : l'une des lignes ne compile pas. Il faut indiquer laquelle.

```
1 public static void
2       main(String[] args){
3 A a=new A(); B b=new B();
4 A ab = new B();
5 a.method(1);
6 a.method(1.);
7 a.method(a);
8 a.method(b);
9 a.method(ab);
10 b.method(a);
11 b.method(b);
12 b.method(ab);
13 ab.method(a);
14 ab.method(b);
15 ab.method(ab);
```

1 pt pour les colonnes 1, 3, 4
0.5 pt pour la colonne 2, la plus facile
on arrondit au demi point supérieur.

5	②	7	① ②	10	③ ②	13	③ ②
6	Erreur compil	8	① ⑤	11	④ ⑤	14	③ ⑤
		9	① ⑤	12	③ ⑤	15	③ ⑤

Exercice 3 – Vecteur et exceptions (12 pts)

On va créer une classe `Vecteur`, fondée sur deux attributs réels, ne permettant pas de sortir de l'intervalle [0, 10]. Toute sortie de l'intervalle lèvera une exception.

Q 3.1 (2pts) Donner le code de la classe `VecteurHorsLimiteException` qui étend les exceptions et contient un constructeur avec un argument `double d`. Lorsqu'une valeur `d` ne sera pas admissible, l'exception enverra le message : `"La valeur ... n'est pas admissible"`

```
1 public class VecteurHorsLimiteException extends Exception {
2     public VecteurHorsLimiteException(double d) {
3         super("La␣valeur␣"+d+ "␣n'est␣pas␣admissible");
4     }
5 }
```

Q 3.2 Donner le code d'une classe `Vecteur` dont les deux attributs réels doivent être compris entre 0 et 10 inclus. Tout dépassement sera sanctionné d'une exception `VecteurHorsLimiteException`.

Q 3.2.1 (2pts) Commencer par donner le code de la méthode privée `testBound` (dont la signature est volontairement non fournie) testant si les attributs sont admissibles et levant une exception dans le cas contraire avec un message sur l'attribut défaillant et sa valeur.
Expliquer en une phrase pourquoi cette méthode est `private`.

Q 3.2.2 (3pts) Ajouter un constructeur à deux arguments, une méthode `toString()` et une méthode d'addition retournant un nouveau vecteur.

Q 3.2.3 (2pts) Ajouter une méthode `equals` dans la classe.

Q 3.3 (3pts) Donner le code d'un programme de test :

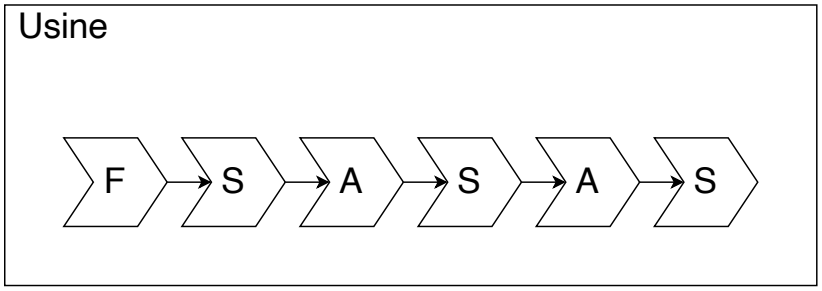
- On créera un vecteur conforme et un non conforme et on testera l'addition entre 2 vecteurs.
- Le `main` ne doit jamais planter sur une `VecteurHorsLimiteException` : toutes les exceptions de ce type seront attrapées.
- Lorsqu'une exception est attrapée, toutes les instances de `Vecteur` seront affichées.
  Note : bien réfléchir à la logique de blocs.
- le main doit afficher un message *tout s'est bien passé* seulement dans le cas où il n'y a pas eu d'exception levée.

```
1 package exam.interro3_2015;
2
3 public class Vecteur {
4     private double x,y;
5
6     public Vecteur(double x, double y) throws VecteurHorsLimiteException{
7         this.x = x;
8         this.y = y;
9         testBound();
10    }
11
12    private void testBound() throws VecteurHorsLimiteException{
13        if(x<0 || x>10)
14            throw new VecteurHorsLimiteException("x out: "+x);
15        if(y<0 || y>10)
16            throw new VecteurHorsLimiteException("y out: "+y);
17    }
18
19    public String toString() {
20        return "Vecteur [x=" + x + ", y=" + y + "]";
21    }
22
23    public Vecteur addition(Vecteur v) throws VecteurHorsLimiteException{
24        Vecteur retour = new Vecteur(x+v.x,y+v.y); // test integre
25        return retour;
26    }
27
28    public boolean equals(Object o){
29        if(o==null) return false;
30        if(o==this) return true;
31        if(o.getClass() != getClass()) return false;
32        Vecteur v = (Vecteur) o;
33        return v.x == this.x && v.y == this.y;
34    }
35
36    public static void main(String[] args) {
37        Vecteur v1=null, v2=null, v3=null; // declaration en dehors du try/catch, sinon
              affichage impossible
38        try{
39            v1 = new Vecteur(1,2);
40            //v2 = new Vecteur(4,11); // test 1
41            //v2 = new Vecteur(4,9); // test 2
42            v2 = new Vecteur(4,5); // test 3
43            v3 = v1.addition(v2);
44            System.out.println("Toutes les operations se sont bien passees !");
45        }catch(VecteurHorsLimiteException e){
46            System.out.println(e.getMessage());
47            System.out.println(v1+ " "+ v2+" "+v3);
48        }
```

```
49
50          System.out.println("tout␣est␣OK␣maintenant␣!"); // OPT
51
52      }
53
54 }
```

Exercice 4 – Optimisation logistique (20.5 pts)

On s'intéresse dans cet exercice à l'optimisation de la logistique dans une usine par une approche de simulation.
Comme le montre la figure, une **usine** est composée d'une alternance d'**unités** : des **fournisseurs** (F), du **stockage** (S) ou des **ateliers** (A) de transformation. Toutes les unités sont susceptibles de tomber en panne. L'originalité de l'architecture vient de l'introduction d'un design pattern composite pour l'**usine**, qui sera elle-même une **unité**.
L'enjeu de la chaîne de production est de transformer des **produits** : un produit **A** rentre dans l'usine, les ateliers le transforme en **B** puis en **C**. Ces produits seront organisés hiérarchiquement.
On ne modélise pas les acheteurs, l'idée est donc de mesurer la vitesse à laquelle nous sommes capable de constituer un stock en sortie d'usine et le coût associé.



Chaque **unité** de l'**usine** sera dérivée d'une interface unique permettant :

- déclencher une `void action()`
- calculer des coûts :
  - de construction `double getCoutInitial()`
  - de fonctionnement, à chaque pas de temps `double getCoutFonct()`
  - d'en cours, pour calculer la valeur des stocks intermédiaires en en fin de cycle, `double getCoutFinal()`.

Note : on ne s'occupe pas des `import` dans cet exercice

Q 4.1 (1.5pt) Donner le diagramme de classe simplifié sur les précédemment éléments mentionnés en gras.

```
1 Unité
2     Usine (composé d'unités)
3 ␣␣␣␣fournisseur
4 ␣␣␣␣stockage
5 ␣␣␣␣atelier␣(utilise␣des␣produits)
6
7 Produit
8 ␣␣␣␣A
9 ␣␣␣␣B
10 ␣␣␣␣C
11
12 Les␣produits␣sont␣utilisés␣par␣les␣unités
```

Q 4.2 (1pt) Donner le code de l'interface `Unite`.

```java
public interface Unite{
    public void action();
    public double getCoutInitial();
    public double getCoutFonct();
    public double getCoutFinal();
}
```

Q 4.3 (5pts) Donner le code de la classe `Stockage`, qui étend la classe `ArrayList` sur les objets `Produit` (tout en respectant la hiérarchie établie précédemment). Les unités de stockage possèdent :

- un constructeur qui initialise la `capacite` et le `coutUnitaireInitial` à partir des arguments ; les attributs seront `public` mais immutables.
- Une méthode d'ajout et une méthode de retrait de `Produit`.
  - La méthode `get` retournera le produit d'indice 0 (en utilisant la méthode `remove(0)` de la classe `ArrayList`). Si le stock est vide, la méthode retourne `null` et affiche un message dans la console.
  - La méthode `put` ajoutera un produit dans le stock (méthode `add(Produit p)` de la classe `ArrayList`) et retournera `true` en cas de succès. Si le stock est plein, on n'effectue pas l'opération mais on envoie un message dans la console et en renvoie `false`.
- les méthodes imposées par l'héritage avec les instructions suivantes :
  - la méthode action ne fait rien.
  - Le coût initial correspond à la capacité multipliée par le cout unitaire.
  - Le coût de fonctionnement correspond à 1% du coût initial.
    Note : afin de rendre le code plus élégant, la valeur de 1% sera stockée dans une constante.
  - Le coût final correspond à la valeur du stock (tous les produits ont une méthode `double getCout()`.

1pt pour extends + implements corrects + constructeur
2pts get+put : if + bons appels à super + gestion OK des returns
1pt pour les couts (pas de pénalité en cas de pb sur action)
0.5pt pour la bonne gestion de la constante (en majuscules !)
0.5pt pour les attributs public final

```java
import java.util.ArrayList;

public class Stockage extends ArrayList<Produit> implements Unite {

    public final static double RATIOCOUTFONCTIONNEMENT = 0.01;
    public final int capacite;
    public final double coutInitialUnitaire;


    public Stockage(int capacite, double coutInitialUnitaire) {
        super();
        this.capacite = capacite;
        this.coutInitialUnitaire = coutInitialUnitaire;
    }


    public Produit get() {
        if(super.size()>0){
            if(super.size()<0.1*capacite)
                System.out.println("Stock à reconstituer");
            return super.remove(0);
        }
        else
            System.out.println("Stockage plein");
        return null;
    }

    public boolean put(Produit p) {
        if(super.size()<this.capacite){
```

```
30              super.add(p);
31              return true;
32          }
33          else{
34              System.out.println("Stockage plein");
35              return false;
36          }
37
38      }
39
40
41      @Override
42      public void action() {
43      }
44
45      @Override
46      public double getCoutInitial() {
47          // TODO Auto-generated method stub
48          return coutInitialUnitaire*capacite;
49      }
50
51      @Override
52      public double getCoutFonct() {
53          // TODO Auto-generated method stub
54          return RATIOCOUTFONCTIONNEMENT*coutInitialUnitaire*capacite;
55      }
56
57      @Override
58      public double getCoutFinal() {
59          double tot = 0;
60          for(Produit p:this)
61              tot += p.getCout();
62          return tot;
63      }
64  }
```

Q 4.4 (5pts) Donner le code de la classe `Atelier` qui permet de transformer des produit selon le processus suivant :

- la classe possède des liens vers les `Stockage precedent` et `suivant` ; un `coutConstruction` réel ; une `capacite` entière (=nb de produits transformés en une itération) ; `tauxDePanne` compris entre 0. et 1. ; un booléen `fonctionnel`, initialisé à `true` qui indique si l'atelier est fonctionnel (ou en panne).
- L'action de l'atelier consiste à :
  - vérifier si l'atelier est `fonctionnel`. Si ce n'est pas le cas, remettre le booléen à `true` et envoyer un message dans la console indiquant la réparation.
  - effectuer les opérations suivantes dans une boucle (nb itérations= `capacite`) :
    - récupérer un `Produit` dans le Stockage précédent ;
    - le transformer le produit en utilisant la méthode static `Produit transform(Produit p)` de la classe `Transformer` (qui ne sera pas étudiée dans cet examen) ;
    - mettre ce `Produit` dans le Stockage suivant ;
    - détecter une éventuelle panne avec un test sur un nombre aléatoire (procédure classique régulièrement vue en TP) ;

    Note : en cas de Stock précédent vide, stock suivant plein ou panne, vous lèverez une `RuntimeException`. L'ensemble de la boucle sera effectuée dans un try/catch permettant de basculer le booléen `fonctionnel` à `false` en cas de problème.
- Le coût initial correspond directement à l'attribut `coutConstruction` ;
- le coût de fonctionnement vaut 1% du coût de construction en temps normal et 2% en cas de panne ;
- le coût final est nul.

```
1
2 1 pt pour déclaration + constructeur (c'est fastidieux...)
3
4 1 pt pour la forme générale d'action (if + get + put)
5
6 0.5pt pour le bon appel à Transformer
7
8 1.5pt pour la gestion des exceptions
9
10 1pt pour les couts
11
12 public class Atelier implements Unite {
13
14     private Stockage prev, next;
15     private boolean fonctionnel;
16     private double txPanne;
17     private double coutInitial;
18     private int capacite;
19
20
21     public Atelier(Stockage prev, Stockage next, double txPanne, double coutInitial, int
           capacite) {
22         this.prev = prev;
23         this.next = next;
24         this.fonctionnel = true;
25         this.txPanne = txPanne;
26         this.coutInitial = coutInitial;
27         this.capacite = capacite;
28     }
29
30
31     @Override
32     public void action() {
33         if(fonctionnel){
34             try{
35                 for(int i =0; i<capacite; i++){
36                     Produit p = prev.get();
37                     if(p == null) throw new RuntimeException("plus de stock");
38                     boolean b = next.put(Transformer.transform(p));
39                     if(!b) throw new RuntimeException("plus de place après");
40                     if(Math.random()<txPanne) throw new RuntimeException("plus de place après")
                           ;
41                 }
42             } catch(Exception e){
43                 fonctionnel = false;
44             }
45         }
46         else{
47             System.out.println("réparation...");
48         }
49
50     }
51     @Override
52     public double getCoutInitial() {
53         return coutInitial;
54     }
55     @Override
56     public double getCoutFonct() {
57         return fonctionnel?0.01*coutInitial:0.02*coutInitial;
58     }
59     @Override
60     public double getCoutFinal() {
61         return 0;
62     }
63 }
```

Q 4.5 (1.5pt) Les `Fournisseur` sont des `Unite` simples : ils ont pour attribut un `Stockage entrée`, initialisé dans le constructeur. Ils n'ont pas de coût (tout est inclus dans les `Produit`). Afin de gagner du temps, vous ne donnerez pas le code des fonctions de cout. La méthode `action` consiste à vérifier le stock de l'usine et à le remplir complètement

de `Produit A` dès qu'il passe sous les 10%.
Donner le code de la classe `Fournisseur`.

1.5pt pour la méthode action avec la bonne interaction avec Stockage

```
public class Fournisseur implements Unite {

    private Stockage next;

    public Fournisseur(Stockage next) {
        this.next = next;
    }

    @Override
    public void action() {
        if(next.size() < next.capacite * 0.1){
            while(next.size() < next.capacite)
                next.put(new A(1)));
        }

    }

    // OPT (non demandé dans l'exercice)
    @Override
    public double getCoutInitial() {
        return 0;
    }
    @Override
    public double getCoutFonct() {
        return 0;
    }
    @Override
    public double getCoutFinal() {
        return 0;
    }




}
```

Q 4.6 (2.5pts) L'`Usine` est construite selon le design pattern composite : il s'agit d'une `Unite` composée d'une ArrayList d'`Unite`.
Son action consiste à invoquer toutes les méthodes actions des `Unite` qui la composent. Il en va de même pour les calculs des coûts initiaux, finaux et de fonctionnement.
Afin de gagner du temps, on considérera que le constructeur reçoit directement en argument une `ArrayList` d'unités. De plus, vous ne donnerez le code d'une seule fonction de coût (les trois sont identiques).

1pt signature + attribut 1.5 pt action + cout

```
public class Usine implements Unite {

    private ArrayList<Unite> allUnits;

    public Usine(ArrayList<Unite> allUnits) {
        this.allUnits = allUnits;
    }

    @Override
    public void action() {
        for(Unite u: allUnits)
            u.action();

    }
    @Override
```

```
16      public double getCoutInitial() {
17          double tot = 0;
18          for(Unite u: allUnits)
19              tot += u.getCoutInitial();
20
21          return tot;
22      }
23      // OPTION : une seule fonction cout demandée
24      @Override
25      public double getCoutFonct() {
26          double tot = 0;
27          for(Unite u: allUnits)
28              tot += u.getCoutFonct();
29
30          return tot;
31      }
32      @Override
33      public double getCoutFinal() {
34
35         double tot = 0;
36          for(Unite u: allUnits)
37              tot += u.getCoutFinal();
38
39          return tot;
40      }
41  }
```

Q 4.7 (4pts) Donner le code d'un `main` permettant d'initialiser l'usine de la figure puis de la faire fonctionner pendant 10 itérations et enfin d'afficher les coûts correspondants.
Tous les coûts de stockage seront pris à 1 et tous les coûts de construction d'atelier à 100. Les taux de panne de tous les éléments seront fixés à 0.1%.
Les capacités de stockage sont arbitrairement mises à 100 et les capacités des ateliers limitées à 1.
Note : il faut commencer par initialiser les unités de stockage, puis les fournisseurs et les ateliers avant de finir par l'usine.