

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)

```
1 public class Boule {  
2     private String couleur;  
3  
4     public Boule(String couleur) {  
5         this.couleur = couleur;  
6     }  
7 }
```

```
1 public static void main(String[] args){  
2     Boule b1 = new Boule("rouge");  
3     Boule b2 = new Boule("jaune");  
4     Boule b3 = new Boule("bleue");  
5     Boule b4 = b1;  
6 }
```

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` ?

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

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).

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.

```
1 public class BouleRandomFactory {  
2     public final String[] couleurs = {"rouge", "jaune", "bleue", "verte", "orange", "violette"};  
3  
4     public static Boule build(){  
5         return new Boule(couleurs[(int) Math.random() * couleurs.length]);  
6     }  
7 }
```

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).

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("Livre18");  
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 ?

Q 2.2 (2.5pts) On envisage d'ajouter une 26ème ligne de code dans le main. Pour chaque proposition (=une des lignes 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]));

```

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);

```

Q 2.4 (2pts) On souhaite que chaque ouvrage (`Livre` ou `BD`) soit associé à un identifiant entier unique immuable, 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.

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

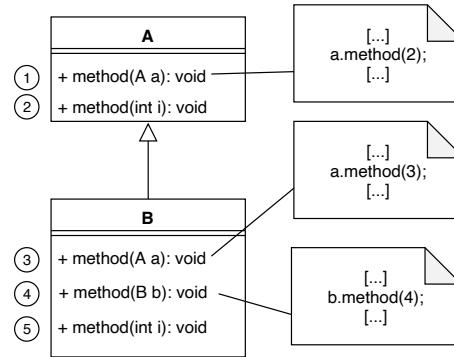
La ligne 7 correspond à une invocation de la fonction ① qui, elle-même invoque la fonction ②

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

```

1 public static void
2     main(String [] args){      5 a . method(1);      7 a . method(a);
3 A a=new A(); B b=new B();       6 a . method(1);      8 a . method(b);
4 A ab = new 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);

```



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"

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.

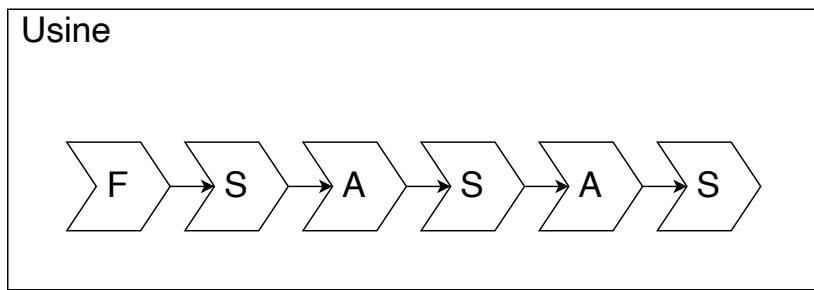
Q 3.3 (3pts) Donner le code d'un programme de test :

- 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.

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 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.

Q 4.2 (1pt) Donner le code de l'interface **Unité**.

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 `getCost()`).

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.

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`.

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).

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.