IN201 : Conception et programmation orientées objet

Examen

Cet examen est composé de 8 exercices indépendants. Le barême indiqué pour chaque exercice l'est à titre indicatif et peut être modifié.
Les seuls documents autorisés pour cet examen sont :

- les cartes de référence distribuées en cours
- une feuille recto-verso de notes manuscrites au format A4

Les annales des examens des années précédentes sont interdites. Les téléphones portables doivent être éteints et rangés. L'utilisation d'un ordinateur durant l'examen est interdite.
Vous avez normalement la place de répondre sur ce document. Si vous devez utiliser une copie libre, n'oubliez pas d'y indiquer vos noms, prénoms et groupe de PC, ainsi que le numéro précis de la question à laquelle vous répondez.
Il est fortement conseillé de lire complétement un exercice avant d'y répondre.

Nom : _____

Prénom : _____

Groupe : _____

Question	Points	Score
1	9	
2	2	
3	2	
4	2	
5	3	
6	2	
7	2	
8	2	
Total :	24	

Commentaires éventuels :

1. Une *file avec priorité* (*priority queue* en anglais) est une structure de données permettant de stocker des éléments comparables entre eux et de récupérer l'élément le plus petit (au sens de la relation de comparaison) avec une complexité constante. On considérera ici que la file ne possède que trois opérations : une opération `add` pour insérer un élément dans la file, une opération `peek` pour enlever et récupérer le plus petit élément de la file et une opération `isEmpty` pour savoir si la file est vide.

Afin de pouvoir comparer des éléments entre eux, on supposera que les données stockées dans la queue ont un type réalisant l'interface `Comparable<T>` fournie par l'API de Java. Cette interface ne possède qu'une méthode dont la signature est `int compareTo(T o)` permettant d'implanter un ordre naturel et dont le fonctionnement est le suivant :

- si **`this`** est plus petit que `o` au sens de l'ordre naturel, alors la méthode renvoie `-1`
- si **`this`** est égal à `o` au sens de l'ordre naturel, alors la méthode renvoie `0`
- si **`this`** est plus grand que `o` au sens de l'ordre naturel, alors la méthode renvoie `1`

On peut implanter une file avec priorité de différentes façons, en utilisant un tableau, en utilisant une liste chaînée etc. On choisira ici de représenter la file avec un *tas binaire* (*binary heap* en anglais). Un tas binaire est une structure de données arborescente respectant les propriétés suivantes :

- chaque nœud a au plus deux fils appelés fils gauche et fils droit
- l'arbre est *complet*, i.e. tous les niveaux de l'arbre excepté éventuellement le dernier sont remplis
- la valeur contenue dans chaque nœud est plus petite que les valeurs de ses deux fils

Par exemple, l'arbre suivant contenant des entiers présenté sur la figure 1 est un tas binaire.

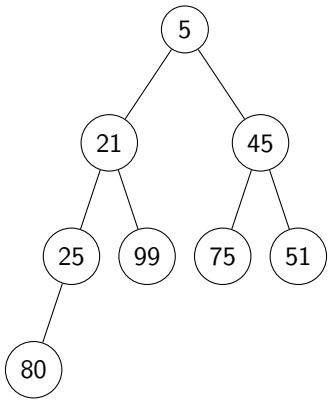
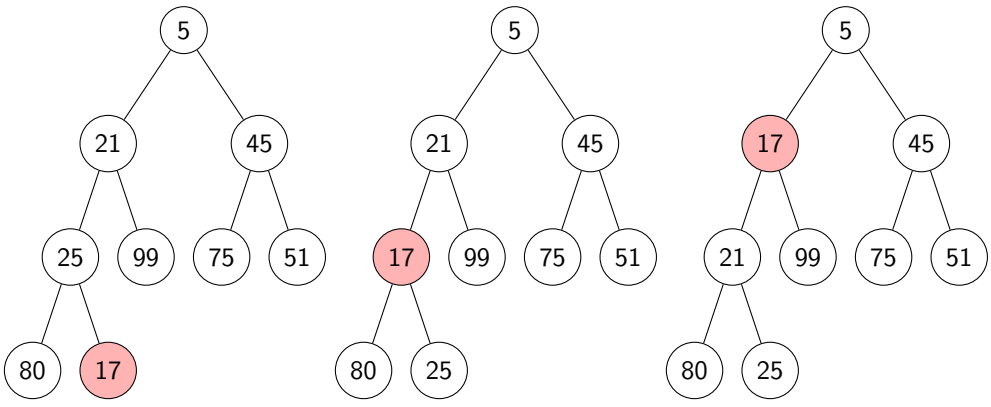


FIGURE 1 – Un tas binaire

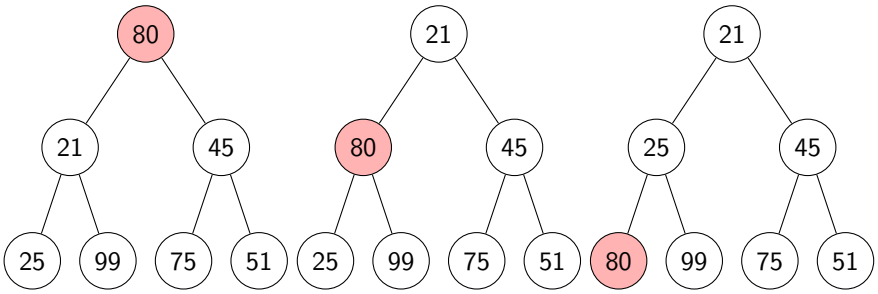
Le tas binaire permet de récupérer la plus petite valeur qu'il contient en temps constant : on sait que cette dernière sera toujours à la racine de l'arbre. Pour garantir cela, les opérations d'insertion et de récupération du plus petit élément sont définies algorithmiquement comme suit :

- pour insérer un élément, on utiliser l'algorithme 1. Celui-ci insère le nouveau nœud en « bas » de l'arbre et fait remonter la nouvelle valeur pour respecter la définition du tas.
  Par exemple, voici les différentes étapes d'insertion de la valeur 17 dans le tas précédent :



- lorsque l'on élimine la racine de l'arbre lors d'un appel à `peek`, on remplace la racine par le dernier

élément de l'arbre (i.e. l'élément le plus à droite dans le dernier niveau) et on fait « descendre » la valeur de la racine pour respecter la définition du tas. L'algorithme 2 sera utilisé.
Par exemple, voici les différentes étapes d'un appel sur peek sur le tas présenté sur la figure 1 (l'appel renverra la valeur 5) :



Pour pouvoir coder ces deux méthodes, il faut pouvoir trouver le premier emplacement disponible et le dernier élément de l'arbre. Pour cela, il suffit d'utiliser un algorithme d'exploration en largeur de l'arbre. L'algorithme 3 présente un algorithme simple renvoyant le premier nœud de l'arbre ayant une place disponible. L'algorithme 4 présente un algorithme renvoyant le dernier nœud de l'arbre et le supprimant de l'arbre.

(½ pt) (a) les structures de données ainsi définies sont-elles génériques ? Si oui, doit-on imposer une borne pour le paramètre formel de type associé ?

(½ pt) (b) doit-on représenter `PriorityQueue` par une classe ? Une classe abstraite ? Une interface ? La réponse sera justifiée.

(2 pt) (c) proposer un diagramme UML présentant les classes ou interfaces `PriorityQueue`, `BinaryHeap` et `Node`, leurs éventuels attributs, leurs méthodes, leur(s) constructeur(s) et les relations existant entre elles. Si vous souhaitez introduire des accesseurs et modifieurs sur des attributs qui sont nombreux, vous pouvez en écrire un seul et mettre « ... » pour les autres. Vous pourrez introduire des méthodes auxiliaires dans les classes en réfléchissant à leur visibilité.

Dans les questions suivantes, vous serez amenés à écrire du code Java. Il n'est pas nécessaire d'écrire la documentation Javadoc des classes, sauf si vous voulez préciser la signification d'un paramètre particulier. Comme précédemment, si vous devez écrire des accesseurs ou des modifieurs triviaux dans une classe, vous pouvez n'en écrire qu'un seul et préciser par un commentaire que les autres accesseurs et modifieurs seront écrits de la même façon.

On rappelle que la classe `ArrayList<E>` possède les méthodes suivantes :

- **`boolean`** `isEmpty()` qui permet de savoir si la liste est vide
- **`void`** `add(E elt)` qui ajoute l'élément `elt` en fin de liste
- `E get(`**`int`**` index)` qui renvoie l'élément situé à l'index `index`
- `E remove(`**`int`**` index)` qui renvoie l'élément situé à l'index `index` et le supprime de la liste

(1 pt) (d) écrire `PriorityQueue` en Java.

(1 pt) (e) écrire `Node` en Java.

(3 pt) (f) écrire `BinaryHeap` en Java.

(1 pt) (g) écrire un programme dans une classe `Main` déclarant une poignée de type`PriorityQueue`, lui affectant le un tas binaire contenant les valeurs du tas présenté sur sur la figure 1 et affichant le plus petit entier contenu dans la file avec priorité.

Algorithme 1 : Ajouter un élément dans le tas binaire

Entrées : l'élément e à ajouter

```
1  trouver le premier nœud p sous lequel on peut ajouter un nœud ;
2  construire un nœud n contenant e et ayant pour père p ;
3  placer le nœud n correctement sous p
4  pere ← p ;
5  noeud ← n ;
6  tant que pere existe faire
7      si la valeur de pere est plus grande que celle de noeud alors
8          échanger les valeurs de pere et de noeud ;
9      si le père de pere n'existe pas alors
10         retourner ;
11     noeud ← pere ;
12     pere ← le père de noeud ;
```

Algorithme 2 : Récupérer la plus petite valeur contenue dans le tas

Sorties : la plus petite valeur contenue dans le tas

```
1  valeur ← la valeur contenue dans la racine du tas ;
2  si la racine n'a pas de fils alors
3      retourner valeur ;
4  donner comme valeur à la racine la valeur du dernier nœud du tas ;
5  noeud ← la racine du tas ;
6  tant que (noeud a un fils gauche de valeur plus petite que noeud) ou (noeud a un fils droit de valeur plus
   petite que noeud) faire
7      si noeud n'a pas de fils droit ou si la valeur du fils gauche de noeud est plus petite que celle de son fils
       droit alors
8          échanger les valeurs de noeud avec celle de son fils gauche ;
9          noeud ← le fils gauche de noeud ;
10     sinon
11         échanger les valeurs de noeud avec celle de son fils droit ;
12         noeud ← le fils droit de noeud ;
13 retourner valeur ;
```

Algorithme 3 : Premier emplacement disponible

Entrées : le nœud racine de l'arbre

Sorties : le premier nœud de l'arbre ayant une place disponible

```
1  construire une liste ln contenant le nœud racine ;
2  tant que ln n'est pas vide faire
3      n ← premier nœud dans ln ;
4      supprimer premier nœud dans ln ;
5      si le fils gauche de n n'existe pas alors
6          retourner n ;
7      ajouter le fils gauche de n à la fin de ln ;
8      si le fils droit de n n'existe pas alors
9          retourner n ;
10     ajouter le fils droit de n à la fin de ln ;
11 retourner null ;
```

Algorithme 4 : Dernier nœud de l'arbre et suppression de ce nœud

Entrées : le nœud racine de l'arbre
Sorties : le dernier nœud de l'arbre (il est également supprimé)

```
1  construire une liste ln contenant le nœud racine ;
2  tant que ln n'est pas vide faire
3  |  n ← premier nœud dans ln ;
4  |  supprimer premier nœud dans ln ;
5  |  si le fils gauche de n existe alors
6  |  |  ajouter le fils gauche de n à la fin de ln ;
7  |  si le fils droit de n existe alors
8  |  |  ajouter le fils droit de n à la fin de ln ;
9  np ← nœud père de n ;
10 si np n'existe pas alors
11 |  la racine de l'arbre ← null ;
12 sinon si le fils gauche de np est n alors
13 |  supprimer le fils gauche de np
14 sinon
15 |  supprimer le fils droit de np
16 retourner n ;
```

2. On dispose d'une classe `Group` modélisant un groupe au sens mathématique du terme. On rappelle qu'un groupe défini sur un ensemble E avec une loi interne $\times$ possède les propriétés suivantes :

- il existe $e \in E$ appelé élément neutre vérifiant la propriété suivante : $\forall x \in E,\ e \times x = x \times e = x$
- la loi $\times$ est associative : $\forall x_1 \in E\ \forall x_2 \in E\ \forall x_3 \in E,\ x_1 \times (x_2 \times x_3) = (x_1 \times x_2) \times x_3$
- tout élement $x \in E$ possède un inverse noté x^{-1} vérifiant $x \times x^{-1} = x^{-1} \times x = e$

Le diagramme de classe de `Group` est présenté sur la figure 3. La classe `E` est définie par ailleurs et la méthode `getElement` permet de récupérer au hasard un élément de E différent de l'élément neutre. La méthode `times` correspond à $\times$ et la méthode `inv` renvoie l'inverse d'un élement de E.

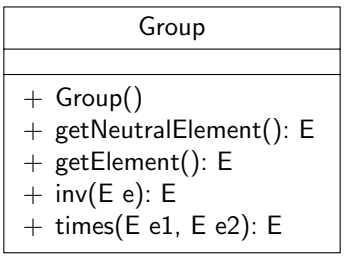


FIGURE 3 – La classe `Group`

(1 pt) (a) écrire une classe de test JUnit permettant de tester si les propriétés mathématique du groupe sont vérifiées dans la classe `Group`. On ne testera les propriétés que sur une instance.

(1 pt) (b) on suppose que l'on définit une classe `AbelianGroup` héritant de `Group`. La classe `AbelianGroup` redéfinit la méthode `times`. D'après le principe de substitution, une instance de `AbelianGroup` doit se comporter de la même façon qu'une instance de `Group`. Comment écrire simplement la classe de test de `AbelianGroup` pour pouvoir rejouer les tests définis dans `GroupTest` sans les réécrire ? On indiquera les éventuelles modifications à apporter à la classe `GroupTest`.

(2 pt) **3.** On s'intéresse ici aux phénomènes se passant dans le LHC (*Large Hydron Collider*), l'accélérateur de particules du CERN. On considère le scénario suivant d'utilisation du LHC : le LHC crée deux protons et les accélère en leur donnant des facteurs d'accélération opposés. Au bout d'un certain temps, le LHC va déclencher la collision des deux protons en les injectant dans la boucle principale de l'accélérateur. Lors de la collision, représentée par un appel de méthode du LHC sur le premier proton, les deux protons vont émettre deux bosons Z^0 qui vont se combiner pour donner naissance à un boson de Higgs.

Représenter le scénario précédent par un diagramme de séquence.

4. On considère les deux classes `A` et `B` suivantes et le programme applicatif défini dans la classe `Main` :

```
1  class A {
2
3      protected void m(double d) {
4          System.out.println("appel de la methode m de A, valeur : " + d);
5      }
6  }
7
8  class B extends A {
9
10     @Override protected void m(double d) {
11         System.out.println("appel de la methode m de B (double), valeur : " + d);
12     }
13
14     protected void m(int i) {
15         System.out.println("appel de la methode m de B (int), valeur : " + i);
16     }
17 }
18
19 class Main {
20     public static void main(String[] args) {
21         A obj = new B();
22
23         obj.m(2);
24     }
25 }
```

(1 pt) (a) est-ce que les trois classes précédentes compilent ? Si non, quelles sont les erreurs détectées par les compilateur ? Comment les corriger ?

(1 pt) (b) les éventuelles erreurs dans les classes étant corrigées, quel est le résultat de l'exécution de `Main` ?

(3 pt) **5.** On cherche à fournir à un centre de recherche en biologie un logiciel de simulation de réaction du système immunitaire humain à différents virus et bactéries. Dans un premier temps, on cherche donc à proposer un modèle objet des notions « métier » nécessaires aux biologistes pour modéliser le système immunitaire.

Le système immunitaire est une collection de mécanismes permettant à un organisme de se protéger de l'infection en détectant et en tuant des agents pathogènes.

Le système immunitaire peut être décomposé en trois couches : une couche physique, le système inné et le système adaptatif. **On ne s'intéressera ici qu'au deux dernières**.

Le système immunitaire inné entre en action lorsqu'un organisme étranger entre dans le corps. Il propose une réponse générique au pathogène sous plusieurs formes : inflammation, système du complément (ensemble d'une vingtaine de protéines permettant de détruire la membrane des cellules étrangères), leucocytes (globules blancs). Ces derniers ont plusieurs formes, la plus connue étant les phagocytes, tous capables de phagocyter un pathogène. On trouve parmi eux les macrophages, les neutrophiles et les cellules dendritiques.

Le système immunitaire adaptatif fournit lui une réponse spécialisée par type de pathogène. Ses principales fonctions sont : la reconnaissance des antigènes n'appartenant pas au corps, la génération d'une réponse adaptée via la production de lymphocytes et le développement d'une mémoire immunologique. Les lymphocytes sont un type particulier de leucocytes et peuvent reconnaître des cibles pathogènes particulières. Ils se divisent en deux catégories : les lymphocytes T, et les lymphocytes B, qui peuvent créer des plasmocytes fabriquant des anticorps.

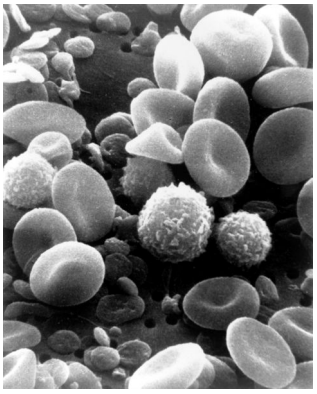


FIGURE 4 – Une photo prise au MEB d'un échantillon de sang circulant humain. On distingue de nombreux globules rouges sous forme de disques applatis biconcaves, ainsi que de nombreux types de globules blancs : des lymphocytes, un monocyte, un neutrophile, et de nombreuses plaquettes (source NCA, 1982).

Proposer un diagramme de classes d'analyse représentant le domaine étudié. On fera apparaître les classes, les relations entre classes, les noms de rôles et les multiplicités des associations, et on justifiera l'utilisation de classes abstraites et d'interfaces. On représentera également les méthodes importantes identifiées dans le texte.

6. Passionné de zoologie, vous avez décidé de construire une application Java permettant de représenter différents types d'animaux. Vous avez commencé par vos animaux préférés, les canards, en créant une interface `Duck` possédant deux méthodes :

– **`void`** `quack()` qui permet de faire caquetter le canard
– `walk(`**`double`** `distance)` qui permet de faire marcher le canard sur une certaine distance

Vous avez ensuite implanté plusieurs réalisations de cette interface : la classe `MallardDuck`, la classe `WoodDuck` etc.

En avançant dans la construction de votre application, vous avez défini une interface `Animal` plus générale avec les méthodes suivantes :

– **`void`** `makeNoise()`
– `move(`**`double`** `distance)` qui permet de faire bouger l'animal en question

Vous allez maintenant pouvoir gérer des collections complètes d'animaux variés. Malheureusement, le disque dur de votre vieil ordinateur vous a lâché et vous ne disposez plus que du *bytecode* de l'interface `Duck` et de ses réalisations. Impossible donc de modifier ces classes et vous ne pouvez pas les réécrire, les caquettements des canards vous ayant demandé trop de temps à implanter...

On veut pouvoir considérer ici des `MallardDuck`, `WoodDuck` etc. comme des objets de type `Animal`. On pourrait par exemple les insérer dans une liste de `Animal` et lorsque l'on demande à tous les objets de la liste de faire du bruit, les canards caquetteront.

(½ pt) (a) en ne modifiant pas les classes et interfaces existantes, quel est le mécanisme le plus simple (vu en cours) permettant de créer des canards pouvant être utilisés dans une liste d'objets de type `Animal` ? Cette solution est-elle pertinente ? Pourquoi ?

(½ pt) (b) pour pallier le problème précédent, on peut utiliser un patron conception appelé *adaptateur*. L'idée de l'adaptateur est de pouvoir utiliser un objet défini avec une interface particulière via une autre interface (il s'agit d'une analogie avec les adaptateurs pour les différents types de prises de courant suivant les pays). Un diagramme de classe présentant l'adaptateur est donné sur la figure 6. Dans ce patron de conception, on souhaite utiliser des instances de `Adaptee` comme des instances de `Standard` et faire en sorte que la méthode `m2` corresponde à la méthode `m1`. La classe `Adapter` permet de faire cela.

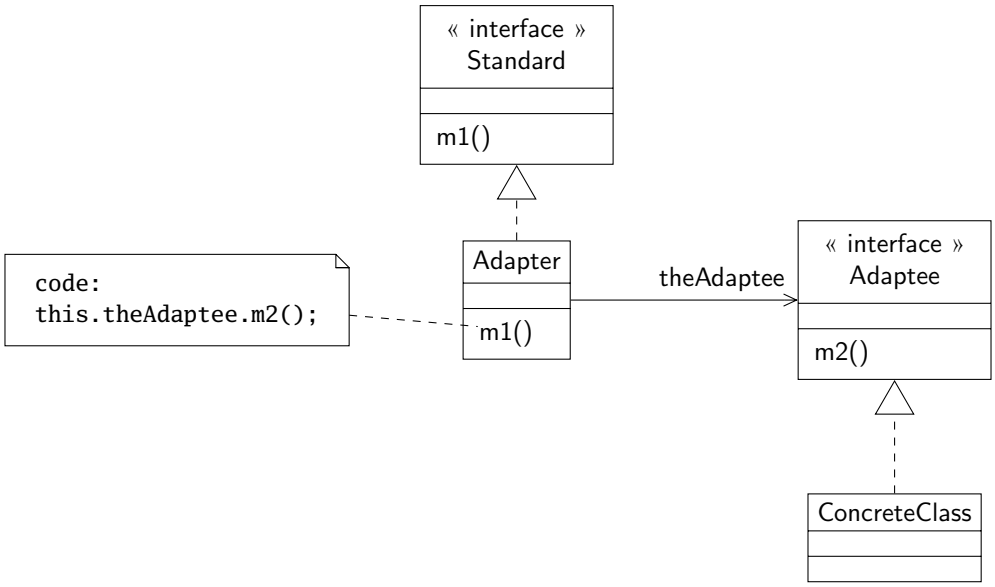


FIGURE 6 – Le patron de conception adaptateur

Adapter le patron de conception adaptateur au problème posé en proposant un diagramme de classes. La classe représentant l'adaptateur sera appelée `DuckAdapter`.

(1 pt) (c) écrire la classe `DuckAdapter` en Java.

(2 pt) **7.** Soient les classes `PrixHT` et `PrixTTC` présentées sur les listings suivants :

```
 1 public class PrixHT {
 2
 3     double valeur;
 4
 5     public PrixHT() {
 6         this.valeur = 0;
 7     }
 8
 9     public PrixHT(double valeur) {
10         this.valeur = valeur;
11     }
12
13     public PrixHT ajouter(PrixHT prix) {
14         return new PrixHT(this.valeur + prix.valeur);
15     }
16 }
```

```
 1 public class PrixTTC extends PrixHT {
 2
 3     double tauxTVA;
 4
 5     public PrixTTC(double tauxTVA) {
 6         this.tauxTVA = tauxTVA;
 7     }
 8
 9     public PrixTTC(double valeur, double tauxTVA) {
10         this.tauxTVA = tauxTVA;
11         super(valeur);
12     }
13
14     @Override public PrixTTC ajouter(PrixTTC prix) {
15         return new PrixTTC(this.valeur + prix.valeur, this.tauxTVA);
16     }
17 }
```

Quelles sont les deux erreurs dans ces classes qui seront détectées par le compilateur Java ? Quelles sont les deux principes énoncés en cours non respectés dans ce code ? On pourra soit indiquer **en rouge** directement dans le code source les erreurs ou se servir des numéros de ligne pour les préciser. On expliquera brièvement les erreurs.

(2 pt) **8.** On considère une classe `A` possédant une méthode `m` renvoyant une valeur de type **`double`** et pouvant lever une exception de type `MyException`. On souhaite implanter une méthode `sum` ayant les caractéristiques suivantes :

- elle prend en paramètre une liste d'objets de type `A` et un entier `nbMax`
- elle renvoie la somme des valeurs renvoyées par `m` appliquée sur les éléments de la liste en ne tenant pas compte des éventuels éléments de type `A` sur lesquels un appel à `m` lève une exception de type `MyException`
- si plus de `nbMax` objets contenus dans la liste lèvent une exception de type `MyException` lors de l'appel à `m`, la méthode `sum` lève une nouvelle exception de type `MyException`

Écrire la méthode `sum`.