Examen de compléments en POO (session 2)

Mercredi 20 juin

— Durée : 2h45
— Épreuve à réaliser sans document à l'exception d'une feuille A4 manuscrite recto-verso.
— Répondre à l'exercice 1 directement sur le sujet, à joindre au reste du rendu. Le sujet entier sera de toute façon mis à disposition sur Moodle après l'épreuve.

I) Questions de cours

Notation de l'exercice 1 :

— *La notation est établie de telle sorte que la stratégie consistant à répondre au hasard de façon équiprobable ait une espérance de note proche de* 0.
— *Pour cette raison, répondre de façon incorrecte à une proposition retire des points...*
— *... mais* **vous avez le droit de ne pas répondre** *! (dans ce cas, pas de pénalité)*
— *En pratique, vous évaluerez chaque proposition en inscrivant "V"(rai), "F"(aux) ou rien (pour s'abstenir) dans la case correspondante.*
— *Un coefficient de difficulté sera déterminé pour homogénéiser la difficulté entre les exercices.*
— *La note d'un exercice de QCM est donnée par l'algorithme suivant :*

```
p = nombre de bonnes réponses
p = p - nombre de mauvaises réponses
p = p 𝔼 coefficient de difficulté
p = max(p, 0) // pas de note négative
p = min(p, nombre de propositions) // on ne dépasse pas le barème de l'exercice
retourner p
```

Exercice 1 : QCM (à détacher du sujet et à remettre avec le reste du rendu)

1. ☐ La classe d'un objet donné est connue et interrogeable à l'exécution.
2. ☐ Le type des éléments qu'une instance de tableau a le droit de contenir est connu et interrogeable à l'exécution.
3. ☐ Le type des éléments qu'une instance d'`ArrayList` a le droit de contenir est connu et interrogeable à l'exécution.
4. ☐ La conversion de **int** vers **float** ne perd pas d'information.
5. ☐ `ArrayList<Integer>` est sous-type de `List<Integer>`.
6. ☐ `ArrayList<Integer>` est sous-type de `ArrayList<Object>`.
7. ☐ Une interface définit un type, sous-type de `Object`.
8. ☐ On peut programmer une **enum** qui soit sous-type de `Number` (`Number` est une classe).
9. ☐ Une classe peut avoir plusieurs sous-classes directes.

10. ☐ Une interface peut avoir une classe membre.

11. ☐ Lorsque dans une même classe, on définit deux méthodes de même nom, de même nombre et types de paramètres, mais avec un type de retour différent, alors cette classe compile et ces deux méthodes sont dites “surchargées”.

12. ☐ Une classe **final** peut contenir une méthode **abstract**.

13. ☐ Une classe **abstract** peut contenir une méthode **final**.

14. ☐ Toute classe a un constructeur par défaut ne prenant pas de paramètre.

15. ☐ Java est plus ancien que C.

16. ☐ La commande `java` lance l'exécution de la JVM.

17. ☐ Il est possible qu'à un moment de l'exécution, suite à une désallocation, une adresse non-**null** contenue dans une variable ne pointe plus sur un objet existant.

18. ☐ Le programme suivant on affiche toujours `30` :

```
1  public class C {
2      public static int x = 0;
3      public static void nIncr(int n) { for (int i = 0; i <n ; i++) x++; }
4      public static void main() throws InterruptedException {
5          Thread t1 = new Thread(() -> nIncr(10));
6          Thread t2 = new Thread(() -> nIncr(20));
7          t1.start(); t2.start();
8          t1.join(); t2.join();
9          System.out.println(x);
10     }
11 }
```

19. ☐ Toute exception pouvant être déclenchée dans une méthode doit, soit être rattrapée par un **catch** dans cette méthode, soit être déclarée dans la signature de la méthode via la clause **throws**.

20. ☐ Lorsqu'un programme est compilé, le compilateur détermine, pour chaque occurrence d'appel à une méthode statique, quelle méthode exacte sera appelée à l'exécution (y compris si plusieurs méthodes du même nom existent).

II) Calculatrice booléenne

a) Du contexte

Calculatrices RPN : Une calculatrice RPN (notation polonaise inversée) est une calculatrice dans laquelle les calculs sont entrés en ordre post-fixe (d'abord les opérandes, puis l'opération).

Le programme se présente comme une boucle d'entrée/évaluation/affichage (REPL), dans laquelle à chaque itération, l'utilisateur peut entrer un symbole (opérande ou opération).

L'état de la calculatrice consiste en une pile (LIFO) d'opérandes. Quand un symbole est entré, la calculatrice l'évalue (si simple opérande, le résultat est l'opérande elle-même, si opération : la calculatrice dépile le nombre nécessaire d'opérandes et leur applique l'opération), empile le résultat obtenu et affiche la pile.

Algèbres de Boole : Une algèbre de Boole est, en toute généralité, une structure algébrique sur un ensemble B (pas forcément juste $\{\text{VRAI}, \text{FAUX}\}$), donnée par les propriétés suivantes :

- B contient au moins deux éléments distincts $\bot$ (élément minimal) et $\top$ (élément maximal)
- B est munie de deux opérations internes binaires $\vee$ (conjonction) et $\wedge$ (disjonction) et d'une opération interne unaire $\neg$ (complémentation).
- Une série de propriétés fondamentales (commutativité des opérations binaires, distributivité réciproque, associativité, ...) ... mais totalement inutiles pour l'exercice.

Par exemple, pour un ensemble X, l'ensemble $B = \mathcal{P}(X)$ (des sous-ensembles de X), muni des opérations union, intersection et complémentaire dans X, forme une algèbre de Boole.

L'interface `Deque` : Une pile LIFO peut être représentée par une instance de l'interface `Deque<E>` (par exemple en instanciant la classe `LinkedList<E>`). Cette interface contient en particulier les méthodes suivante :

- `void addFirst(E e)` : insère e en première position de la file/pile.
- `E removeFirst()` : supprime et retourne l'élément en première position de la file/pile.
- `int size()` : (héritée de `Collection<E>`) retourne le nombre déléments en pile.

De plus, `Deque<E>` est sous-type de `Iterable<E>`, donc une instance peut être itérée par une boucle *for-each*. La spécification de `Deque<E>` garantit que l'ordre des éléments est alors respecté.

L'interface `Set` et la classe `HashSet` : L'interface `Set<E>` contient les méthodes suivantes :

- `boolean addAll(Set<E> s)` : ajoute à l'ensemble `this` les éléments de `s`.
- `boolean removeAll(Set<E> s)` : retire de `this` les éléments de `s`.
- `boolean retainAll(Set<E> s)` : retire de `this` les éléments qui ne sont pas dans `s`.

Si vous vous souvenez des autres méthodes, vous avez le droit de les utiliser.

La classe `HashSet<E>` est une implémentation de `Set<E>` possédant un constructeur sans paramètre, pour instancier un ensemble vide.

Enfin, la méthode `toString` de `HashSet` retourne une chaîne de la forme "`[3, 1, 15]`" pour l'ensemble $\{3, 1, 15\}$. On peut donc afficher les éléments d'un ensemble sans l'itérer à la main.

b) Exercices

Exercice 2 : Calculatrice de vérité

Écrivez un programme de calculatrice en RPN qui gère juste les constantes `VRAI` et `FAUX` et les opérations `ET`, `OU` et `NON` (i.e. : l'algèbre de Boole la plus simple, celle des valeurs de vérité).

Exemple d'exécution pour la calculatrice booléenne (le symbole "`>`" est l'invite du REPL, ce qui suit ce symbole est l'entrée de l'utilisateur) :

```
> VRAI
Pile : | VRAI |

> FAUX
Pile : | VRAI | FAUX |

> OU
```

```
Pile : | VRAI |

> ET
Erreur : pas assez d'opérandes.
Pile : | VRAI |

> NON
```

```
Pile : | FAUX |

> 42
Erreur : entrée incorrecte.
Pile : | FAUX |

>
```

Remarques : un programme qui afficherait toujours la pile à l'envers est aussi considéré correct. Par ailleurs, on préférera le programme le plus clair et le plus simple possible.

Exercice 3 : Algèbres de Boole généralisée

On modélise la notion mathématique d'algèbre de Boole (générale) par une interface Java :

```java
public interface BooleanAlgebra<T> {
    T minElement(); // élément minimal de l'algèbre (constante)
    T maxElement(); // élément maximal de l'algèbre (constante)
    T meet(T x, T y); // première opération binaire (conjonction)
    T join(T x, T y); // deuxième opération binaire (disjonction)
    T complement(T x); // opération unaire (complémentation)
}
```

Remarques :

- *Une instance de cette interface représente l'algèbre elle-même, i.e. un objet par lequel on accède aux opérations d'une algèbre donnée. Elle ne représente pas un élément de l'algèbre. Ainsi, les opérations binaires prennent bien 2 paramètres dans `T` (2 éléments de l'algèbre).*
- *Pour une calculatrice donnée, une seule instance de `BooleanAlgebra` est nécessaire.*
- *Les implémentations des opérations de `BooleanAlgebra<T>` ne modifient pas les objets passés en paramètre. Si une nouvelle valeur doit être retournée, alors une nouvelle instance de `T` est créée.*

Questions/à faire :

1. Écrivez une classe `TruthAlgebra` qui implémente l'interface `BooleanAlgebra` pour les valeurs de vérité (représentables par le type **boolean** ou `Boolean`).

   Ici `meet` est l'opération `ET`, `join` l'opération `OU`, `complement` l'opération `NON`, `minElement` est la constante `FAUX` et `maxElement` est la constante `VRAI`.

2. Dans une classe `TestTruthAlgebra`, écrivez une méthode `main` qui effectue l'opération `(NON (FAUX ET VRAI)) OU FAUX` à l'aide de `TruthAlgebra` et qui affiche son résultat.

   Écrivez en commentaire les affichages attendus.

3. Écrivez une classe `SubsetAlgebra<E>` qui implémente l'interface `BooleanAlgebra` pour les sous-ensembles d'un certain ensemble X fini et non vide (X sera un paramètre, de type `Set<E>`, passé au constructeur).

   - *Ici `meet` est l'intersection, `join` l'union, `complement` le complément d'un ensemble dans X, `minElement` est l'ensemble vide et `maxElement` l'ensemble X.*

— *Le type des éléments de X n'est pas fixé dans cette implémentation. Ainsi `SubsetAlgebra` a un paramètre de type, `E`, pour le type des éléments des ensembles manipulés.* **Attention, `T` (dans `BooleanAlgebra<T>`) et `E` ne jouent pas le même rôle !**

4. Dans une classe `TestSubsetAlgebra`, écrivez une méthode `main` qui effectue l'opération $\overline{\{1,2,3\} \cup \{2,4\}}$ (notation : $\overline{F}$ = "complément de F") à l'aide de la classe `SubsetAlgebra` et affiche son résultat. Les opérandes sont des sous-ensembles de $X = \{0,1,2,3,4,5\}$. Écrivez en commentaire les affichages attendus.

Exercice 4 :

On programme maintenant une calculatrice RPN réutilisable pour toute algèbre de Boole.

Pour l'interface utilisateur, il nous manque encore les méthodes pour convertir une chaîne de caractères en symboles de l'algèbre et vice-versa. Malheureusement, leur implémentation est spécifique à chaque algèbre (les éléments et les opérations y sont à chaque fois représentés différemment). On suppose donc ces méthodes fournies par une instance de l'interface suivante :

```
1  public interface BooleIO<T> {
2      enum Operator { MEET, JOIN, COMPLEMENT; }
3
4      /** Si input est la représentation d'une valeur de l'algèbre, retourne celle-ci, sinon retourne
           null. */
5      T parseElement(String input);
6      /** Retourne une représentation en chaîne de l'élément en paramètre */
7      String toString(T e);
8      /** Si input est la représentation d'un opérateur, retourne celui-ci, sinon retourne null. */
9      Operator parseOperator(String input);
10     /** Retourne une représentation en chaîne de l'opérateur en paramètre */
11     String toString(Operator o);
12 }
```

À faire : Écrire la calculatrice (classe `BACalculator<T>`). Celle-ci utilise une implémentation de l'algèbre de Boole et un convertisseur de/vers `String` (instances, respectivement, de `BooleanAlgebra<T>` et de `BooleIO<T>`, pour le même `T`) passés en paramètre du constructeur de `BACalculator`. Cette classe a une méthode `launch` qui contient la boucle d'exécution (REPL).

III) Concurrence

Rappels pour la programmation concurrente :

Dans la classe `Thread` :

— Constructeur `Thread()` : instancie un `Thread` dont la méthode `run()` ne fait rien.

— **`void start()`** : démarre un nouveau *thread*, associé à l'instance de `Thread` sur laquelle elle est appelée, en y exécutant la méthode `run()` de celle-ci.

— **`void run()`** : contient les instructions que le *thread* devra exécuter, à savoir : rien par défaut. À redéfinir quand on étend `Thread`.

Dans la classe `Object` :

— **`void wait()`** **`throws InterruptedException`** : met le *thread* courant en attente d'une notification sur le moniteur de l'objet. Typiquement utilisé dans une boucle **`while`**.

— **`void notifyAll()`** : envoie une notification à tous les *threads* en attente sur le moniteur de l'objet.

Ces 2 méthodes doivent être appelées dans un bloc synchronisé sur l'objet récepteur de l'appel.

Par ailleurs, on rappelle que le moniteur d'un objet Java est un verrou réentrant : un *thread* qui tente d'acquérir un moniteur qu'il possède déjà n'est pas bloqué. De plus, le moniteur ne sera libéré qu'en sortant du bloc **`synchronized`** le plus externe.

Exercice 5 : Marchandage

Dans un site web de vente d'objets de particulier à particulier, on veut implémenter un mécanisme permettant à acheteur et vendeur de négocier un prix pour un objet donné.

Une négociation s'effectue par une succession de propositions de prix de la part de l'acheteur et du vendeur (pas forcément à tour de rôle). La négociation est conclue dès lors qu'acheteur et vendeur s'accordent sur le prix. Alors, la transaction peut être effectuée par la plateforme.

La négociation sera modélisée par une instance de l'interface suivante :

```
1  public interface Negociation {
2      int getPropositionAcheteur();
3      void setPropositionAcheteur(int prop);
4      int getPropositionVendeur();
5      void setPropositionVendeur(int prop);
6      boolean estConclue();
7      void effectue();
8  }
```

Le contrat de cette interface oblige ses implémentations à

— être *thread safe* : leurs méthodes peuvent être exécutées dans des *threads* différents sans remettre en cause leur bon fonctionnement ;

— dans les 2 méthodes `setProposition`, notifier tous les *threads* en attente sur moniteur de l'instance courante de `Negociation` (raison : une négociation contient typiquement des phases d'attente d'une proposition de l'autre partie) ;

— empêcher qu'une nouvelle proposition soit faite quand l'affaire est déjà conclue.

1. Écrire la classe `NegociationImpl` implémentant l'interface `Negociation` et telle que :
   — la méthode `effectue` se contente d'afficher le prix de vente final si l'affaire est conclue ou bien un message d'erreur sinon ;
   — la classe `NegociationImpl` ait un constructeur fixant le prix initial de l'acheteur à 0 et celui du vendeur à la valeur passée en paramètre.
2. Écrire la classe `AcheteurDetermine` étendant `Thread`, avec un constructeur prenant en paramètre une négociation, et dont la méthode `run` décrit le comportement suivant :
   Initialement, l'acheteur fait une proposition (comprise entre 0 et le prix de vente initial), puis tant que la négociation n'est pas conclue, il "attend" une proposition du vendeur et fait une nouvelle proposition (strictement supérieure à sa dernière proposition, inférieure à la dernière du vendeur).
3. Écrire la classe `VendeurDetermine` étendant `Thread`, avec un constructeur prenant en paramètre une négociation, et dont la méthode `run` décrit le comportement suivant :
   D'abord, si la proposition actuelle de l'acheteur est strictement supérieure à 0, le vendeur fait une proposition ; puis tant que la négociation n'est pas conclue, il "attend" une proposition de l'acheteur et fait une nouvelle proposition (strictement inférieure à sa dernière proposition, supérieure à la dernière de l'acheteur).
4. Écrire la classe `TestNegociation` contenant une méthode `main` qui instancie une négociation, lance un *thread* pour l'acheteur et un pour le vendeur, puis "attend" que le marchandage se termine "effectuer" la transaction.
5. Question : pourquoi le *thread* vendeur doit-il d'abord faire une proposition, avant d'attendre, si la proposition acheteur est strictement supérieure à 0 au début ?