UAMOB
Département d'informatique
Niveau **M1 ISIL+M1 GSI**, Jul.**2021,**
Corrigé d'Examen de rattrapage **MAL + ASSI**
Durée : 1H :15m

Exercice 1 : (20 points)

Lors du cours, lorsque nous voulions manipuler un ensemble d'objets, nous avons fait appel à des *collections*. En Java, Les collections sont un ensemble de classes appartenant au paquetage `java.util` et représentant différentes façons de stocker un ensemble d'objets : listes, tas, arbres etc. Nous considérerons dans toute la suite que toutes les classes et interfaces appartiennent à `java.util`. Il faudra en tenir compte lors de l'écriture du code.

Pour pouvoir accéder au contenu d'une collection, nous avons utilisé des itérateurs : un itérateur est un objet permettant d'accéder de façon uniforme aux objets d'une collection, sans s'occuper de la façon dont ces objets sont stockés. Un itérateur est un objet implantant l'interface `java.util.Iterator`. Cette interface possède trois méthodes :

- `hasNext()` qui renvoie un booléen. Elle renvoie **`true`** si la collection possède encore des éléments non parcourus ;
- `next()` qui renvoie le prochain élément de la collection sous la forme d'une instance d'`Object` et qui avance d'un élément dans la collection ;
- `remove()` qui enlève l'élément courant.

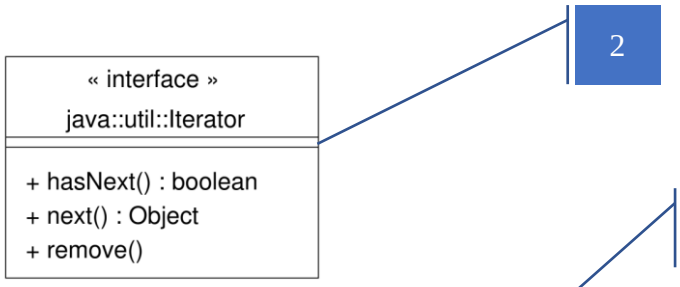
1. représenter l'interface `Iterator` sur un diagramme UML ;
2. écrire en Java une méthode statique `affiche` qui prend un itérateur en paramètre et affiche toutes les chaînes de caractères se trouvant dans la collection correspondante ;
3. les anciennes classes représentant les collections en Java (par exemple `Vector` ou `Stack`) implantent une méthode `elements` qui renvoie un objet de type `Enumeration`. L'interface `Enumeration` permet également de parcourir ces collections, mais ne permet pas d'effacer un élément de la collection. Ses méthodes sont :
   - `hasMoreElements()` qui renvoie **`true`** si il y a encore des éléments à parcourir dans la collection ;
   - `nextElement()` qui renvoie le prochain élément sous la forme d'une instance d'`Object`.

   Représenter l'interface sur le diagramme UML précédent.
4. on dispose d'un objet de type `Enumeration` sur une instance de `Vector` par exemple et on souhaiterait pouvoir utiliser dessus la méthode `affiche` développée précédemment sans modifier les interfaces `Enumeration` et `Iterator`, ni la méthode `affiche(Iterator)` développée précédemment. Est-ce possible ? Pourquoi ? Quel « mécanisme » intervient ?
5. pour résoudre ce problème, nous allons utiliser un *design pattern* : l'adaptateur. Il s'agit de réaliser l'interface `Iterator` par une classe `EnumerationIterator` (qui possédera donc toutes les méthodes de l'interface). `EnumerationIterator` utilisera l'instance de `Enumeration` que nous voulions utiliser : les méthodes de `EnumerationIterator` utiliseront celles de `Enumeration`.

   Compléter le diagramme UML avec la classe `EnumerationIterator`.
6. écrire la classe `EnumerationIterator` en Java. On constate que la méthode `remove` n'existe pas dans l'interface `Enumeration`. Indiquer comment écrire la méthode `remove` de `EnumerationIterator`.
7. proposer un diagramme de classe général pour le *design pattern*. L'objectif général de ce *pattern* est d'adapter une interface existante pour l'utiliser dans un nouveau contexte, c'est-à-dire avec une nouvelle interface.

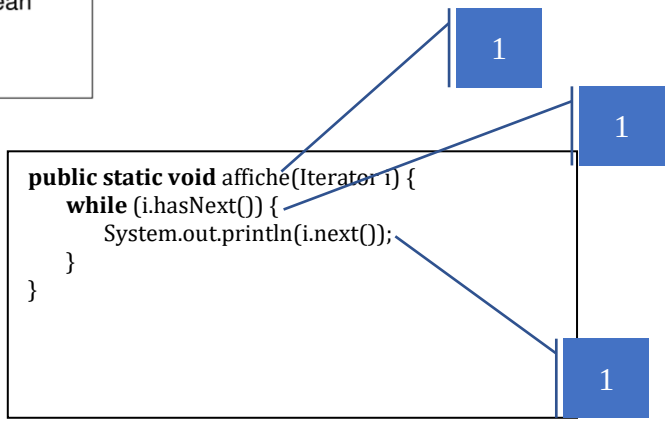
Solution :

Q1) interface Iterator :

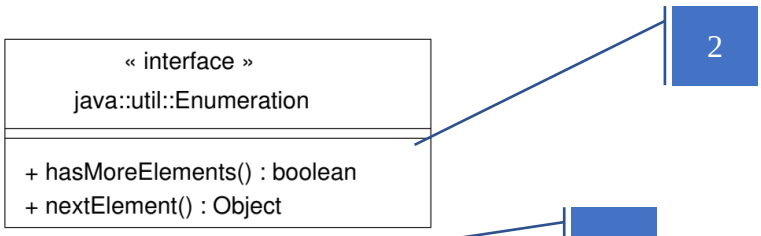


Q2) methode affiche():

```
public static void affiche(java.util.Iterator i) {
        Object o = null;
        while (i.hasNext()) {
                if ((o = i.next()) instanceof String) {
                System.out.println(o);
        }
        }
}
```

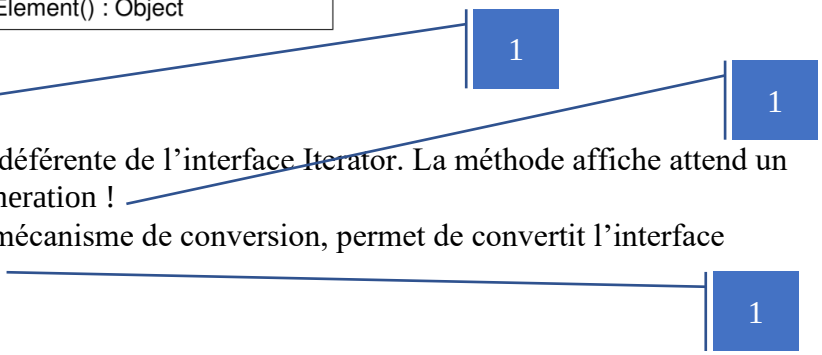


Q3) interface Enumeration



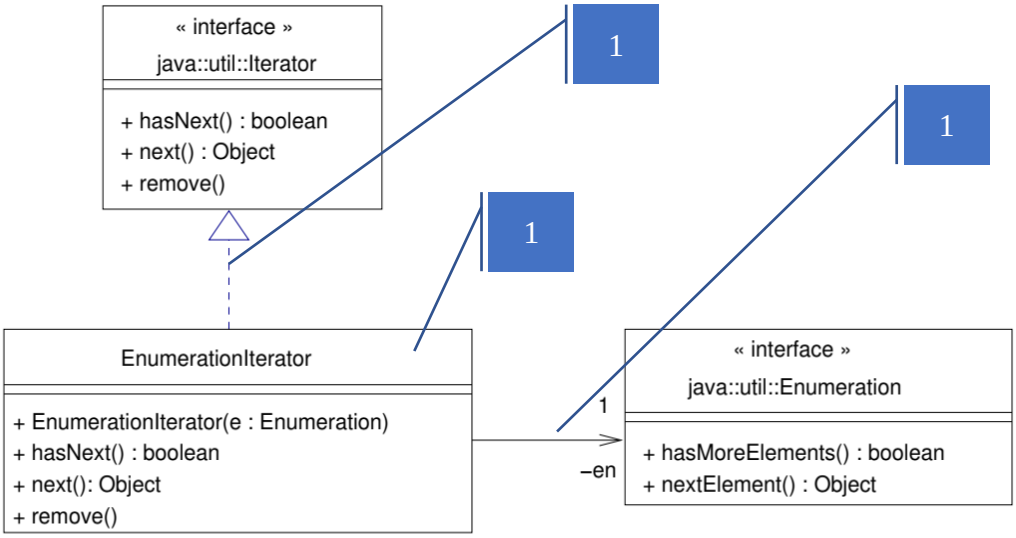
Q4)

- Ce n'est pas possible.
- Parce que l'interface Enumeration est déférente de l'interface Iterator. La méthode affiche attend un Iterator mais nous lui passant un Enumeration !
- Le mécanisme qui intervient c'est un mécanisme de conversion, permet de convertit l'interface Enumeration en interface Iterator.



Q5)

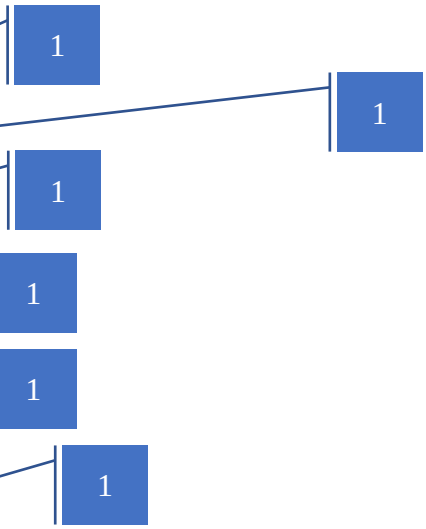
la classe EnumerationIterator réalise l'interface Iterator, et délègue le traitement des méthodes à Enumeration (sens de l'association). Il ne fallait pas oublier le constructeur de la classe.



Q6) La classe EnumerationIterator :

On ne peut pas écrire remove en utilisant Enumeration, car on ne dispose pas d'une méthode similaire. Comme EnumerationIterator realize Iterator, elle doit respecter son contrat.

```
import java.util.Enumeration;
import java.util.Iterator;

public class EnumerationIterator implements Iterator {

        private Enumeration en;

        public EnumerationIterator(Enumeration en) {
                this.en = en;
        }

        // Implementation of java.util.Iterator
        public final boolean hasNext() {
                return this.en.hasMoreElements();
        }

        public final Object next() {
                return this.en.nextElement();
        }

        public final void remove() {
                throw new UnsupportedOperationException();
                //ou bien tout simplment afficher un message « operation non autorisée » ...
        }
}
```



Q7) Diagramme de classe général

La classe Adaptateur devait : Réaliser l'interface cible pour que le principe de substitution puisse s'appliquer ; déléguer ses appels à l'interface à adapter.