SCIENCES SORBONNE UNIVERSITÉ

LU2IN002-2022oct
Éléments de programmation
par objets avec Java

Examen du 12 janvier 2023 – Durée : 2 heures

Seul document autorisé : **feuille A4 manuscrite, recto-verso.**
Pas de calculatrice ou téléphone. Barème indicatif sur 45.

Partie 1 Exercices (18 points)

Exercice 1 (3 points) (Question des Quiz) On considère les 2 interfaces et la classe suivantes :

```
1  public interface APlume { }
2
3  public interface Colore {
4      public String getCouleur();
5  }
```

```
6  public class Stylo {
7      private int taille;
8      public Stylo(int taille) {
9          this.taille=taille;
10     }
11     public String toString() {
12         return "Stylo "+taille+" cm";
13     }
14 }
```

Un stylo plume coloré a une couleur. Écrire la classe `StyloPlumeColore` avec notamment un constructeur à deux paramètres et une méthode `toString()` qui pour un stylo `"bleu"` de 20 cm doit retourner exactement `"Stylo 20 cm de couleur bleu à plume"`.

```
1  public class StyloPlumeColore extends Stylo implements APlume, Colore {
2      private String couleur;
3      public StyloPlumeColore(String couleur, int taille) {
4          super(taille);
5          this.couleur=couleur;
6      }
7      public String getCouleur() {
8          return couleur;
9      }
10     public String toString() {
11         return super.toString()+" de couleur "+couleur+" à plume";
12     }
13 }
```

Exercice 2 (9 points) On considère la hiérarchie de classes et d'interface suivante.

```
1  public interface Visseur {
2     public void visser();
3  }
4  public abstract class Outil { }
5  public class Marteau extends Outil { }
6  public class Tournevis extends Outil implements Visseur {
7     public void visser() { System.out.println("Le tournevis visse"); }
8  }
9  public abstract class Machine extends Outil {
10    public void agir() { System.out.println("La machine agit"); }
11 }
12 public class Perceuse extends Machine implements Visseur {
13    public void agir() { System.out.println("La perceuse perce"); }
14    public void visser() { System.out.println("La perceuse visse"); }
15 }
```

Q2.1 (2 points) Pour chacune des 4 instructions ci-dessous, écrivez le numéro de la ligne, puis indiquez si la ligne compile (OK) ou pas (FAUX).

```
20 Outil o1 = new Tournevis();
21 Outil o2 = new Machine();
22 Visseur v1 = new Visseur();
23 Visseur v2 = new Tournevis();
```

```
20 Outil o1 = new Tournevis(); // OK
21 Outil o2 = new Machine(); // Faux : on ne peut pas instancier une classe abstraite
22 Visseur v1 = new Visseur(); // Faux : on ne peut pas instancier une interface
23 Visseur v2 = new Tournevis(); // OK
```

Q2.2 (2 points) On considère l'instruction correcte suivante : `Outil op=new Perceuse();`
Pour chacune des 4 instructions suivantes, écrivez le numéro de la ligne, puis indiquez si l'instruction est correcte (OK) ou bien si elle provoque une erreur à la compilation (COMPIL) ou à l'exécution (EXEC).

```
30 Machine mc = op;
31 Visseur vi = (Visseur)op;
32 Perceuse pc = (Machine)op;
33 Marteau mt = (Marteau)op;
```

```
30 Machine mc=op;//Erreur COMPIL : Outil cannot be converted to Machine
31 Visseur vi=(Visseur)op;//OK
32 Perceuse pc=(Machine)op;//Erreur COMPIL : Machine cannot be converted to Perceuse
33 Marteau mt=(Marteau)op;//Erreur EXEC : ClassCastException : Perceuse cannot be cast to Marteau
```

Q2.3 (3 points) On considère l'instruction correcte suivante : `Machine mac=new Perceuse();`
Pour chacune des 6 instructions suivantes, écrivez le numéro de la ligne, puis donnez l'affichage obtenu.

```
40 System.out.println(mac instanceof Perceuse);
41 System.out.println(mac instanceof Outil);
42 System.out.println(mac instanceof Visseur);
43 mac.agir();
44 ((Perceuse)mac).agir();
45 ((Visseur)mac).visser();
```

```
40 System.out.println(mac instanceof Perceuse); // true
41 System.out.println(mac instanceof Outil); // true
42 System.out.println(mac instanceof Visseur); //true
43 mac.agir(); // La perceuse perce (dépend de l'instance)
44 ((Perceuse)mac).agir(); // La perceuse perce (le cast est inutile)
45 ((Visseur)mac).visser(); // La perceuse visse (le cast est obligatoire)
```

Q2.4 (2 points) On veut modéliser une trousse à outils. Donner la déclaration d'un tableau `trousse` initialisé avec un marteau, un tournevis et une perceuse, puis écrire une boucle `for` sans indice pour parcourir tous les outils de la trousse et appeler la méthode `visser()` des outils qui peuvent visser.

```
60 Outil [] trousse={new Marteau(),new Tournevis(),new Perceuse()};
61
62 for(Outil ot : trousse) {
63     if (ot instanceof Visseur) {
64         ((Visseur)ot).visser();
65     }
66 }
```

Exercice 3 (6 points) On considère les deux classes suivantes :

```
1  public class Billet {
2      private int valeur;
3      public Billet(int valeur) {
4          this.valeur=valeur;
5      }
6      public int getValeur() {
7          return valeur;
8      }
9  }
```

```
10 public class FauxBillet extends Billet {
11     public FauxBillet(int valeur) {
12         super(valeur);
13     }
14 }
```

Q3.1 (1 point) Écrire la classe `FauxBilletException` qui contient un constructeur prenant en paramètre un billet. Le message de l'exception doit être par exemple : `"Le billet de 10 euros est faux"`.

```
1  public class FauxBilletException extends Exception {
2      public FauxBilletException(Billet b) {
3          super("Le billet de "+b.getValeur()+" euros est faux");
4      }
5  }
```

Q3.2 (2,5 point) Dans une classe `Detecteur`, écrire la méthode statique `verifier` qui prend en paramètre un billet et qui retourne la valeur du billet. Cette méthode lève l'exception `FauxBilletException` si le billet en paramètre est un faux billet.

```
public class Detecteur {
    public static int verifier(Billet b) throws FauxBilletException {
        if (b instanceof FauxBillet) {
            throw new FauxBilletException(b);
        }
        return b.getValeur();
    }
}
```

Q3.3 (2,5 point) Dans la classe `Detecteur`, ajouter la méthode statique `compter` qui prend en paramètre un tableau de billets et qui retourne la somme des valeurs des billets qui sont vrais (le tableau en paramètre est supposé plein de billets). Cette méthode appelle la méthode `verifier` pour chaque billet et affiche le message de l'exception si le billet est faux. Exemple d'utilisation et d'affichage obtenu :

```
Billet [] tab={ new Billet(5),
                new FauxBillet(10),
                new Billet(20) };
System.out.println("Somme="+Detecteur.compter(tab));
```

Affichage :

Le billet de 10 euros est faux
Somme=25

```
    public static int compter(Billet [] tab) {
        int somme=0;
        for(Billet b : tab) {
            try {
                somme+=verifier(b);
            } catch(FauxBilletException e) {
                System.out.println(e.getMessage());
            }
        }
        return somme;
    }
```

Partie 2 Problème (27 points)

Remarque préliminaire : *la visibilité des variables et des méthodes à définir, ainsi que leurs aspects statique, ou final ne sera pas précisé. C'est à vous de décider de la meilleure solution à apporter. Si la méthode `toString` n'est pas demandée dans la question, il n'est pas nécessaire de la fournir. Sans aucune précision donnée dans la question, une méthode ne rend rien.*

En Python, le type dictionnaire est composé d'un ensemble d'associations. Une association est composée d'une *clé* associée à une *valeur*. Par exemple, le dictionnaire `{(42, "Bonjour"), (11, "le"), (38, "Monde")}` est composé de 3 associations : la 1ère association a pour clé l'entier 42 associée à la valeur `"Bonjour"`, la 2e association a pour clé 11 associée à la valeur `"le"` et la 3e association a pour clé 38 associée à la valeur `"Monde"`. Dans cet exercice, nous allons définir un ensemble de classes pour représenter de tels dictionnaires en Java.

Dans un dictionnaire, deux associations ne peuvent pas avoir la même clé. Ainsi, les clés sont donc des éléments qui doivent donc être comparables afin de garantir leur unicité. Pour cela, dans notre code, une association possèdera une clé de type `Comparable` défini par l'interface suivante :

```
public interface Comparable {
    public boolean estEgal(Object obj);
}
```

Dans une association, la valeur associée à la clé peut être de type quelconque. Dans notre implémentation, une hiérarchie de classes est utilisée pour représenter tout type d'association : la classe `Association` est la classe mère de cette hiérarchie. Dans cette classe, le type de la valeur n'est pas précisé, la classe se décline alors en différentes classes selon le type de la valeur voulue : `AssociationString` pour une association

dont la valeur est de type chaîne de caractères, `AssociationInt` pour une association dont la valeur est de type entier, etc.

Question 4.1 (2 points) En respectant ce qui est dit en préambule, écrire la classe `Association` qui ne comporte qu'un attribut `cle` non modifiable, un constructeur avec un seul argument pour initialiser la clé, et un assesseur `getCle` pour cet attribut. Cette classe contient aussi une méthode `getValeur`, sans argument, qui rend un `Object`, ainsi qu'une méthode `setValeur` qui prend un `Object` en argument. Ces deux dernières méthodes seront définies dans les classes filles (cf. question suivante : la première méthode rendra la valeur de l'association, la seconde modifiera la valeur).

La classe `Association` doit aussi contenir une méthode `toString` qui rend une chaîne fournissant la clé mais aussi la valeur de l'association, le tout mis entre parenthèses, par exemple, la chaîne rendue sera `"(42, Hello)"` pour l'association dont la clé vaut 42 et la valeur est la chaîne "Hello".

Remarque : *dans tout cet exercice, la présence de* `@Override` *n'est pas demandée (c'est donc facultatif dans les solutions attendues donc).*

```
1  public abstract class Association {
2      private final Comparable cle;
3      public Association(Comparable cle) { this.cle = cle; }
4      public Comparable getCle() { return cle; }
5      public abstract Object getValeur();
6      public abstract void setValeur(Object valeur);
7      @Override
8      public String toString() {
9          return "("+cle.toString()+", "+getValeur()+")";
10     }
11 }
```

Question 4.2 (2 points) Écrire la classe `AssociationString` qui étend la classe `Association`. Cette classe contient un unique attribut `valeur` de type `String`, un constructeur à 2 arguments (une clé et une valeur), ainsi que les méthodes `getValeur` et `setValeur`. Pour la méthode `setValeur`, la modification de l'attribut ne se fait que si l'objet donné en argument est du même type que la valeur attendue, sinon la fonction ne fait rien.

```
1  public class AssociationString extends Association {
2      private String valeur;
3      public AssociationString(Comparable cle, String valeur) {
4          super(cle);
5          this.valeur = valeur;
6      }
7
8      @Override
9      public Object getValeur() { return valeur; }
10
11     @Override
12     public void setValeur(Object obj) {
13         if ((obj != null) && (obj.getClass() == valeur.getClass()))
14             valeur = (String)obj;
15
16         // Autre solution possible ici:
17         // if ((obj != null) &&(obj instanceof String))
18         //  valeur = (String)obj;
19     }
20 }
```

Remarque : dans le reste de l'exercice, on considère écrites d'autres classes qui étendent de la même façon la classe `Association`*, par exemple* `AssociationInt`*,* `AssociationChar`*, etc.*

Question 4.3 (2 points) Nous nous concentrons sur des clés à valeur entière. Écrire la classe `CleEntiere` qui n'a qu'un seul attribut de type entier `valCle`. Elle doit posséder un constructeur à un seul argument

pour initialiser l'attribut. Cette classe doit implémenter l'interface `Comparable`. La méthode correspondante doit rendre le booléen vrai si l'objet donné en argument est bien une autre clé de même valeur.

```
1  public class CleEntiere implements Comparable {
2      private int valCle;
3      public CleEntiere(int valeur) {
4          this.valCle = valeur;
5      }
6      @Override
7      public boolean estEgal(Object obj) {
8          if (obj == null) return false;
9          if (obj == this) return true;
10         if (obj.getClass() != this.getClass()) return false;
11         return valCle == ((CleEntiere)obj).valCle;
12     }
13 }
```

Remarque : dans le reste de l'exercice, on considère que l'on a aussi ajouté à cette classe une méthode `toString()` qui rend une chaîne de caractères avec la valeur de la clé.

Question 4.4 (3 points) Dans notre code Java, un dictionnaire est représenté sous la forme d'un tableau dynamique d'associations. Ce tableau dynamique est initialisé à la création du dictionnaire, et on y ajoute les associations, les unes après les autres (l'ajout sera traité dans les questions suivantes).

Écrire la classe `Dictionnaire` qui respecte la spécification ainsi décrite. Cette classe doit contenir un constructeur sans argument, et une méthode `toString` rendant l'ensemble des associations du dictionnaire, encadrées par `{` et `}`. Par exemple, le dictionnaire suivant qui contient 3 associations :
`"{(42, Bonjour), (11, le), (38, Monde)}"`

Ajouter à cette classe la méthode `listeCles()` qui ne prend pas d'argument et rend un tableau dynamique avec seulement les clés des associations présentes dans le dictionnaire. Quel est le type des éléments de ce tableau ?

Remarque : *un tableau dynamique est défini par une `ArrayList`, cf. cours 4. C'est ici préférable car cela évite de devoir préciser une taille maximale pour ce tableau, et permet donc de pouvoir ajouter autant d'associations que nécessaire, sans contrainte de taille.*

```
1  import java.util.ArrayList;
2
3  public class Dictionnaire{
4      private ArrayList<Association> elements;
5      public Dictionnaire() {
6          elements = new ArrayList<Association>();
7      }
8
9      public ArrayList<Comparable> listeCles() {
10         ArrayList<Comparable> res = new ArrayList<Comparable>();
11         for(Association elt : elements) {
12             res.add(elt.getCle());
13         }
14         return res;
15     }
16
17     public String toString() {
18         String res = "{";
19         boolean first = true; // une solution possible parmi d'autres
20         for(Association elt : elements) {
21             if (first)
22                 first = false;
23             else
24                 res += ", ";
```

```
25              res +=  elt.toString();
26          }
27          return res + "}";
28      }
29  }
```

Question 4.5 (2 points) Pour un dictionnaire, il est nécessaire de pouvoir comparer 2 associations afin de déterminer si elles possèdent la même clé. Il faut donc une méthode `teste(Association assoc)` qui rend le booléen vrai si l'association donnée en argument a la même clé que l'association courante.

1. Dans quelle classe doit-on définir cette méthode `teste` ? Pourquoi ?

   Dans la classe `Association` car elle doit être utilisable sans nécessairement connaître le type de la valeur (dans `Dictionnaire` par exemple).

2. Donner le code de la méthode `teste`.

```
1  public boolean teste(Association assoc) {
2      if (assoc == null) return false;  // ne pas oublier ce cas !
3      return cle.estEgal(assoc.getCle());
4  }
```

Question 4.6 (2 points) Dans un dictionnaire, il ne peut pas y avoir 2 associations ayant la même clé. Ainsi, l'ajout d'une association dont la clé est la clé d'une association déjà présente dans le dictionnaire n'a aucun effet (ie. l'association déjà présente est conservée, et la nouvelle association n'est pas retenue).

Écrire la méthode `void ajoute(Association assoc)` de la classe `Dictionnaire` qui permet d'ajouter une association dans le dictionnaire. Si une association existe déjà avec la même clé que l'association donnée en argument, celle-ci n'est pas ajoutée (rien ne se passe).

```
1  public void ajoute(Association assoc) {
2      for(Association elt : elements) {
3          if (elt.teste(assoc))
4              return; // pas d'ajout si déjà dans le dictionnaire
5      }
6      elements.add(assoc);
7  }
```

Question 4.7 (1 point)

Donner les instructions pour créer un dictionnaire de nom `dict` et lui ajouter l'association de clé `42` et de valeur `"Bonjour"`, puis lui ajouter l'association de clé `1138` et de valeur `2023`.

```
1  Dictionnaire dict = new Dictionnaire();
2  dict.ajoute(new AssociationString(new CleEntiere(42),"Bonjour"));
3  dict.ajoute(new AssociationInt(new CleEntiere(1138),2023));
```

Question 4.8 (4 points) On veut ajouter à la classe dictionnaire la possibilité de remplacer la valeur d'une association présente dans le dictionnaire. Cette modification ne peut se faire que si l'association à modifier est présente dans le dictionnaire, dans le cas contraire, une exception doit être levée. Pour cela, on a défini la classe d'exception `CleNonTrouveeException` qui n'a qu'un unique constructeur prenant en argument une chaîne de caractères.

Dans la classe `Dictionnaire`, écrire la méthode `remplace` qui prend en argument une association `assoc` et qui remplace l'association existante dans le dictionnaire ayant la même clé que `assoc` par `assoc`.

Cette méthode lève une exception de type `CleNonTrouveeException` si la modification demandée est impossible. Le message d'erreur qui est associé à la levée de l'exception doit donner la raison pour laquelle la modification est impossible, en particulier, la valeur de la clé non trouvée si c'est le cas.

```
1  public void remplace(Association assoc) throws CleNonTrouveeException {
2      if (assoc == null) throw new CleNonTrouveeException("Association vide!");
```

```
3       for (int i=0; i < elements.size(); i++) {
4           if (elements.get(i).teste(assoc)) {
5               elements.set(i,assoc);
6               // on peut aussi : faire un remove(i) suivi d'un add(assoc)
7               // au lieu de set(i,assoc) car il n'est pas précisé que
8               // l'association doit se trouver obligatoirement à la même place
9               // dans le dictionnaire.
10              return;
11          }
12      }
13      throw new CleNonTrouveeException("Clé non trouvée : " +assoc.getCle());
14  }
```

Question 4.9 (2 points) L'ajout de la méthode `remplace` a un impact dans les fonctions qui vont l'utiliser. Par exemple, on considère une fonction `main` qui utilise un appel à cette fonction. Donner, en les expliquant, les 2 façons d'utiliser cette fonction `remplace`, par exemple pour modifier l'association de clé `42` en lui donnant la nouvelle valeur `"Mondo!"` dans le dictionnaire `dict`.

Deux façons possibles : soit utiliser un bloc `try ... catch`, soit modifier la signature de la fonction `main` pour signaler le passage de l'exception.

Solution try catch :

```
1  try {
2      dict.remplace(new Association(new CleEntiere(42), "Mondo!"));
3  }
4  catch (CleNonTrouveeException e) {
5      System.out.println("Problème de clé..."+e);
6  }
```

Solution main sans try catch :

```
1  public static void main(String[] args) throws CleNonTrouveeException {
2      ...
3      dict.remplace(new Association(new CleEntiere(42), "Mondo!"));
4      ...
5  }
```

Question 4.10 (2 points) On souhaite qu'une `Association` puisse être clonée mais son clonage n'est possible que dans ses classes filles, là où est connue la valeur de l'association. Expliquer comment procéder et donner le code nécessaire pour avoir une fonction `clone()` dans les classes `Association` et `AssociationString`.

Remarque : la clé étant non modifiable et la valeur qui est de type `String` n'ont pas besoin d'être clonées.

Dans la classe `Association` il faut une méthode abstraite (obligatoire pour pouvoir réaliser un appel à `clone` sur une instance de `Association`, dans le dictionnaire par exemple).

```
1      public abstract Association clone();
```

Dans la classe `AssociationString` on implémente la fonction en lui donnant le code de retour qui est soit `Association` soit `AssociationString`, les 2 conviennent.

```
1      public Association clone() {
2          return new AssociationString(super.getCle(), valeur);
3      }
```

Question 4.11 (3 points) On souhaite maintenant pouvoir utiliser des clés en 2 dimensions : c'est-à-dire qui correspondent à des couples avec 2 valeurs entières.

Écrire la classe `Cle2D` qui a pour attribut un tableau de 2 valeurs entières. Cette classe contient un constructeur à 2 arguments entiers et une méthode `toString` qui rend une chaîne composée par les 2 valeurs du tableau encadrées par des parenthèses (notation comme un couple). Cette classe doit permettre

de créer des associations avec des clés à 2 dimensions.

Ici, on connaît la taille du tableau (2 "cases"), cela n'a donc pas de sens de faire un tableau dynamique, un tableau classique est suffisant (moins gourmand en mémoire et traitements plus aisés).

```
1  public class Cle2D implements Comparable {
2      private final int[] vecteur;
3      public Cle2D(int v1, int v2) {
4          vecteur = new int[2];
5          vecteur[0] = v1;
6          vecteur[1] = v2;
7      }
8
9      @Override
10     public boolean estEgal(Object obj) {
11         if (obj == null) return false;
12         if (obj == this) return true;
13         if (obj.getClass() != this.getClass()) return false;
14         Cle2D cle = (Cle2D)obj;
15         return (vecteur[0] == cle.vecteur[0])&&(vecteur[1] == cle.vecteur[1]);
16     }
17
18     @Override
19     public String toString() {
20         return "(" + vecteur[0] +", "+vecteur[1]+")";
21     }
22 }
```

Question 4.12 (2 points) La solution proposée dans cet exercice est critiquable, serait-il possible de réaliser une implémentation d'une telle classe dictionnaire en utilisant les classes génériques ? Si oui, expliquer comment procéder.

Oui, il serait même préférable d'utiliser les classes génériques :

- la hiérarchie des classes `Association` oblige de redéfinir une sous-classe selon le type de la valeur, alors que ces sous-classes font quasiment la même chose. On pourrait donc définir une classe `Association<T>` qui définirait de façon générique le type de la valeur.
- on pourrait aussi remplacer l'utilisation de `Object`, là où c'est possible, par un type générique, ce qui simplifierait l'écriture des classes.
- Cela peut amener à définir un type `T` implémentant l'interface `Comparable` ce qui simplifierait aussi le code des fonctions liées à la clé.