EISTI ING1 - EXAMEN DE PROGRAMMATION JAVA – 1ERE SESSION 2014

Corrigé indicatif

1.1. NON **1.2.** OUI **1.3.** OUI

2.1. Empêcher sa redéfinition dans une classe fille

2.2. Empêcher sa modification après initialisation (à la déclaration ou dans un constructeur)

3.1. - : private / ~ : (vide ; absence de mot clé Java pour package private) / # : protected / + : public

3.2. attributs privés + constructeur en public + ajout accesseurs

3.3.1. vérifier sa non nullité dans le setter et utiliser celui-ci pour toute modification ou initialisation (on peut utiliser Objects.requireNonNull pour cette vérification)

3.3.2. final pour la déclaration, vérification de la non nullité uniquement dans le constructeur et pas de setter

3.3.3. pas de vérification sur la nullité au niveau de la modification ou initialisation mais prise en compte du cas null dans toute méthode utilisant le libellé ; on peut également ajouter un constructeur sans paramètre

3.3.4. oui (s'il n'est pas fourni à l'initialisation, il restera nul)

3.3.5. idem question 3.3.1 et 3.3.2 : dans le setter si modifiable, dans le constructeur sinon

3.4.1. @Override public boolean equals(Object o)

3.4.2.1. this.libellé.equals(comp.libellé) ou équivalent avec les getters

3.4.2.2. Objects.equals(this.libellé, comp.libellé) ou équivalent avec les getters

3.4.3. hashCode (2 objets égaux au sens de equals doivent avoir même hashCode)

4.1. Non car le constructeur par défaut appelle le constructeur sans paramètre de la classe mère et celui-ci n'est pas disponible dans la classe Composant

4.2. oui, principe de la surcharge en Java

4.3.
```
public ComposantPrecieux(String libellé, int prix) {
  super(libellé);
  this.prix = prix;          // ou this.setPrix(prix);  // si setter existe
}
public ComposantPrecieux() { this(null,10); }                    // on peut définir 10 dans une constante
public ComposantPrecieux(String libellé) { this(libellé, 10); }  // private static int PRIX_DEFAULT = 10;
public ComposantPrecieux(int prix) { this(null, prix); }
```

4.4.1. super.toString() + " à " + prix + " euros"

4.4.2. "composant "+ Objects.toString(libellé, "anonyme")

4.4.3. joli / très joli à 100 euros / très joli à 100 euros (liaison tardive pourle 3^{e} cas)

4.5. NON **4.6.** OUI **4.7.** On lève l'exception UnsupportedOperationException

5.1. throw : instruction levant une exception
throws : complète la signature d'une méthode avec les exceptions propagées

5.2. oui dans le cas d'un try-with-resources (Java 7)

5.3.1.
```
// toutes les exceptions sont propagées et tout flux ouvert est bien refermé
public static Personne chercherPersonneByNuméro(String filename) throws IOException, ClassNotFoundException,
        ClassCastException {    // ClassCastException conseillée
    try (InputStream is = new FileInputStream(filename); ObjectInputStream ois = new ObjectInputStream(is)) {
        return (Personne) ois.readObject();
    }
}
// Java 6 : difficile à écrire car obligé de faire du try catch finally  pour gérer la fermeture puis relancer exceptions
// Java 7 : en cas de chaînage des ressources, il est fortement conseillé de déclarer les ressources séparément
// en cas de problème sur le dernier chaînon (ici ObjectInputStream)
```

5.3.2.
```
// aucune exception propagée; null renvoyée en cas de pb et tout flux ouvert est bien refermé
```

```
public static Personne chercherPersonneByNuméro(String filename) {  // version 2 return
    try (InputStream is = new FileInputStream(filename); ObjectInputStream ois = new ObjectInputStream(is)) {
        return (Personne) ois.readObject();
    } catch (IOException | ClassNotFoundException | ClassCastException e) { // ClassCastException obligatoire
        return null;
    }
} // en Java 6 : try catch + finally pour faire le close + try catch dans finally car close peut échouer
public static Personne chercherPersonneByNuméro(String filename) {  // version 1 return
     Personne p;
     try (InputStream is = new FileInputStream(filename); ObjectInputStream ois = new ObjectInputStream(is)) {
        p = (Personne) ois.readObject();
     } catch (IOException | ClassNotFoundException | ClassCastException e) { // ClassCastException obligatoire
        p = null;
     }
     return p;
  }
```

6.1. SortedSet ou NavigableSet **6.2.** List **6.3.** hashCode (puis equals)

6.4. ordre naturel sur les éléments ou compareTo si les éléments sont Comparable ou compare si Comparator sur les éléménts est fourni à la construction du TreeSet

6.5. Map<Integer, Queue<Personne>> filesAttentesParPriorité **6.6.1.** on parcourt Jour.values()

6.6.2.
```
Iterator<Evénement> it = événements.iterator() ;
while (it.hasNext) {
        Evénement e = it.next();
        if (e.getJour() == jour) {
                it.remove() ;
        } // il est interdit de faire événements.remove(e) avec itérateur explicite ou implicite
}
```

7.1.
```
if (cmp ==null || coll==null || coll.isEmpty())  throw new IllegalArgumentException(
            "La médiane sur une collection de valeur est définie à partir d'au moins 1 élément et 1 comparateur");
 List<T> l = new ArrayList<>(coll); Collections.sort(l,cmp);
 return l.get(l.size()/2);   // si la taille est paire renvoie le 1er élément de la 2e moitié
 // SortedSet<T> coll2 = new TreeSet<>(coll,cmp) aurait bien trié les elements mais en éliminant les doublons
// ce n'est pas valable pour des collections qui ne sont pas des ensembles, d'où le choix de la liste pour le tri.
```

7.2. La clause n'est pas obligatoire techniquement car l'exception est de type RuntimeException mais elle est néanmoins fortement conseillée car explicitement levée en cas de problème

7.3. public static <T>T mediane(Collection<? extends T> coll, Comparator<? super T> cmp)

7.4.
```
public class ComparateurEntier implements Comparator<Integer>{
        @Override
        public int compare(Integer o1, Integer o2) {
          int p1 = o1 % 2;  // parité de o1 : 0 = pair / 1 = impair
          int p2 = o2 % 2;  // parité de o2
          int res = p1 - p2;  // pairs inférieurs aux impairs : 0 < 1
          if (res == 0) { // même parité                    // ou :
             if (p1==0) res = o1 - o2; // 2 pairs          // res =o1 - o2;
             else res = o2 - o1; // 2 impairs               // if (p1==1) res = -res;  // ordre inverse pr impairs
          }
          return res;
        }
}
```

7.5.

```
List<Integer> c = new ArrayList<>(); // par exemple
Collections.addAll(c, 1, 2, 3, 4, 5, 6, 7);   // valeurs triées : 2,4,6,7,5,3,1
int m= Utils.mediane(c, new ComparateurEntier());   // la réponse est 7
System.out.println("La médiane de " + c + " est " + m);
```