Examen de Programmation Java

mercredi 5 juin 2013

Examen papier. Durée 2H.
Aucun document sauf une feuille A4 recto-verso.
Pas d'ordinateur.
Sauf précision, la version de Java utilisée est Java SE 7.
On vous donne en annexe des extraits de la documentation Java SE 7.

I - Questions de cours

Q1. Peut-on avoir un constructeur dans une classe abstraite ?

Q2. Peut-on avoir un constructeur dans une interface ?

Q3. Peut-on créer une instance (avec `new`) d'une classe abstraite ?

Q4. Peut-on créer une instance (avec `new`) d'une classe interface ?

Q5. A quoi sert l'annotation `@Override` sur une méthode ?

Q6. En quoi Ant et un IDE (Eclipse, Netbeans ou IntelliJ IDEA) sont-ils complémentaires ?

Q7. Pour chaque instanciation suivante, préciser pour quelle(s) version(s) de Java la ligne de code provoque une erreur de compilation. On se limitera aux versions de Java SE : 4, 6 et 7.

```
a) List<Personne> l1 = new LinkedList<>();
b) List<Personne> l2 = new LinkedList<Personne>();
c) List l3 = new LinkedList();
```

II - Exercices

Exercice 1

```
public class Redef {
   public static void main(String[] args) {
      Y o1= new Y();             // hashCode en hexa : 3fdb8a73
      System.out.println(o1.toString());
      System.out.println(o1.toStr());
      Object o2= new Object();  // hashCode en hexa : 3fdb8a74
      System.out.println(o2.toString());
      o2= new Y();               // hashCode en hexa : 3fdb8a75
      System.out.println(o2.toString());
   }
}

public class Y {
   public String toString() { return "dans Y"; }
   public String toStr() { return super.toString(); }
}
```

Q1. Qu'est-ce qui s'affiche à l'écran ? Justifier votre réponse.

Exercice 2

Soit la classe Garage contenant une collection de voitures. On vous donne uniquement l'entête de cette classe :

```
public class Garage implements Iterable<Voiture> { … }
```

Q1. Quelle méthode doit être implémentée par la classe Garage ? Répondre en donnant l'entête complète de la méthode.

Q2. On considère une classe cliente utilisant la classe garage. Donner le code de la méthode d'affichage complet d'un garage en réutilisant la représentation textuelle de base de chaque voiture. L'entête de la méthode à coder est la suivante :

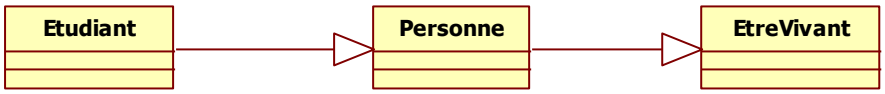
```
private static void afficherGarage(Garage garage) { … }
```

Exercice 3

Dans la classe java.util.Collections, on considère la méthode suivante :

```
public static <T> T min(java.util.Collection<? extends T> coll,
          java.util.Comparator<? super T> comp)
```

On a également la chaîne d'héritage suivante sur les 3 classes suivantes :



Les variables suivantes référencent 3 collections non nulles :

```
List<Etudiant> listeEtudiant;
Set<Personne> ensemblePersonne;
Deque<EtreVivant> pileEtreVivant;
```

Les variables suivantes référencent 3 comparateurs non nuls :

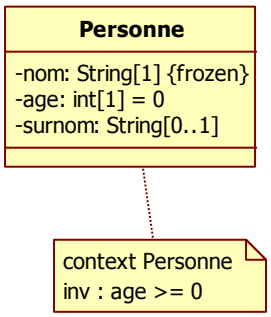
```
Comparator<Etudiant> cmpEtudiant;
Comparator<Personne> cmpPersonne;
Comparator<EtreVivant> cmpEtreVivant;
```

Q1. Donner la liste de tous les appels incorrects (vis-à-vis du compilateur) parmi les suivants ? Justifier votre réponse.

a) `Collections.min(ensemblePersonne, cmpEtudiant)`
b) `Collections.min(ensemblePersonne, cmpPersonne)`
c) `Collections.min(ensemblePersonne, cmpEtreVivant)`
d) `Collections.min(listeEtudiant, cmpPersonne)`
e) `Collections.min(pileEtreVivant, cmpPersonne)`

Exercice 4

On vous donne la représentation UML de la classe Personne :



Q1. Ecrire le code Java de cette classe en tenant compte des contraintes UML et OCL ainsi que des instructions suivantes :

a) un constructeur partiel avec le nom seul

b) un constructeur complet avec nom, âge et surnom
c) les accesseurs (getters et setters) nécessaires
d) redéfinir les méthodes `equals`, `hashCode` et `toString` en tenant compte des 3 attributs

On utilisera l'exception `NullPointerException` ou `IllegalArgumentException` si une contrainte UML ou OCL n'est pas vérifiée.

Exercice 5

Une table de hachage permet de stocker des paires "clef-valeur" dans le but d'uniformiser la complexité des accès en lecture et écriture. On définit une fonction de hachage qui, à toute valeur à stocker passée en paramètre, lui attribue un entier compris entre 0 et une valeur max. Chaque valeur de hachage contient un ensemble d'éléments dans lequel on stocke une structure de deux champs, la clef et la valeur.

Par exemple, pour une valeur de hachage comprise entre 0 et 5, si on a :
- `h("bonjour") == 2`
- `h("salut") == 4`
- `h("coucou") == 2`

Et si on souhaite associer les couples :
- `"bonjour" => "jaune"`
- `"salut" => "vert"`
- `"coucou" => "rouge"`

Alors la table de hachage sera :
```
[0] => {}
[1] => {}
[2] => { ["bonjour","jaune"], ["coucou","rouge"] }
[3] => {}
[4] => { ["salut","vert"] }
[5] => {}
```

En Java, on va hériter de l'interface `eisti.Map<K,V>` , donnée en annexe, pour créer notre table de hachage et de l'interface `java.util.Map.Entry<K,V>` pour gérer les couples stockés.

On vous donne le code de la classe Element et le début du celui de la classe TableHachage :

```
//classe Element
public class Element<K,V> implements java.util.Map.Entry<K,V> {
    private final K key;
    private V value;
    public Element(K key, V value) {
        this.key=key; this.value=value;
    }

    @Override
    public K getKey() { return this.key; }

    @Override
    public V getValue() {  return this.value; }

    @Override
    public V setValue(V value) {
        this.value=value;
        return this.value;
```

```
    }
}
```

//classe TableHachage

```
import java.util.List;
import java.util.ArrayList;
import eisti.Map;

public class TableHachage<K,V> implements Map<K,V> {
    public final static int MAX_HACHAGE=100;

    private final List<List<Element<K,V>>> tableHachage;

    public TableHachage() {
        // création du tableau de MAX_HACHAGE listes vides
        tableHachage = new ArrayList<>(MAX_HACHAGE);
        for (int i=0;i<MAX_HACHAGE;++i) {
            List<Element<K,V>> alveoles = new ArrayList<Element<K,V>>();
            tableHachage.add(alveoles);
        }
    }

    // ramène le hashCode d'un objet dans l'intervalle [0,MAX_HACHAGE[
    private int h(K key) {
        int res=key.hashCode();
        if (res<0) res*=(-1);
        return res%MAX_HACHAGE;
    }

    //TODO : à compléter
}
```

Q1. Donner le code de la méthode `get` de la classe TableHachage.

Q2. Donner le code de la méthode `put` de la classe TableHachage.

Q3. Donner les instructions permettant de créer une table de hachage permettant d'associer un double à une chaîne de caractère.

Annexes

Extrait de l'API de la classe java.lang.Object

`boolean` **`equals`**`(Object obj)`
Indicates whether some other object is "equal to" this one.

`Class<?>` **`getClass`**`()`
Returns the runtime class of this `Object`.

`int` **`hashCode`**`()`
Returns a hash code value for the object.

`String` **`toString`**`()`
Returns a string representation of the object equal to the value of :
`getClass().getName() + '@' + Integer.toHexString(hashCode())`

API de l'interface eisti.Map<K,V>

`void` **`clear`**`()`
Removes all of the mappings from this map (optional operation).

`boolean` **`containsKey`**`(K key)`
Returns `true` if this map contains a mapping for the specified key.

`boolean` **`containsValue`**`(V value)`
Returns `true` if this map maps one or more keys to the specified value.

`Set<Map.Entry<K,V>>` **`entrySet`**`()`
Returns a `Set` view of the mappings contained in this map.

`V` **`get`**`(K key)`
Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.

`boolean` **`isEmpty`**`()`
Returns `true` if this map contains no key-value mappings.

`Set<K>` **`keySet`**`()`
Returns a `Set` view of the keys contained in this map.

`V` **`put`**`(K key, V value)`
Associates the specified value with the specified key in this map.

`void` **`putAll`**`(Map<? extends K,? extends V> m)`
Copies all of the mappings from the specified map to this map.

`V` **`remove`**`(K key)`
Removes the mapping for a key from this map if it is present.

`int` **`size`**`()`
Returns the number of key-value mappings in this map.

`Collection<V>` **`values`**`()`
Returns a `Collection` view of the values contained in this map.

NB :

- les interfaces Set, Map.Entry, Collection sont importées de java.util.
- l'interface eisti.Map<K,V> est une version modifiée de java.util.Map<K,V> en remplaçant les paramètres de type Object par K ou V.

Extrait de l'API de l'interface java.util.List<E>

`boolean` **`add`**`(E e)`
Appends the specified element to the end of this list (optional operation).

`void` **`add`**`(int index, E element)`
Inserts the specified element at the specified position in this list (optional operation).

`boolean` **`addAll`**`(Collection<? extends E> c)`
Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).

`boolean` **`addAll`**`(int index, Collection<? extends E> c)`
Inserts all of the elements in the specified collection into this list at the specified position (optional operation).

`void` **`clear`**`()`
Removes all of the elements from this list (optional operation).

`boolean` **`contains`**`(Object o)`
Returns `true` if this list contains the specified element.

`E` **`get`**`(int index)`
Returns the element at the specified position in this list.

`int` **`indexOf`**`(Object o)`
Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

`boolean` **`isEmpty`**`()`
Returns `true` if this list contains no elements.

`Iterator<E>` **`iterator`**`()`
Returns an iterator over the elements in this list in proper sequence.

`int` **`lastIndexOf`**`(Object o)`
Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.

`E` **`remove`**`(int index)`
Removes the element at the specified position in this list (optional operation).

`boolean` **`remove`**`(Object o)`
Removes the first occurrence of the specified element from this list, if it is present (optional operation).

`boolean` **`removeAll`**`(Collection<?> c)`
Removes from this list all of its elements that are contained in the specified collection (optional operation).

`boolean` **`retainAll`**`(Collection<?> c)`
Retains only the elements in this list that are contained in the specified collection (optional operation).

`E` **`set`**`(int index, E element)`
Replaces the element at the specified position in this list with the specified element (optional operation).

`int` **`size`**`()`
Returns the number of elements in this list.

`List<E>` **`subList`**`(int fromIndex, int toIndex)`
Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`,

exclusive.

API de la classe java.util.Objects

static <T> int	**compare**(T a, T b, Comparator<? super T> c) Returns 0 if the arguments are identical and c.compare(a, b) otherwise.
static boolean	**deepEquals**(Object a, Object b) Returns true if the arguments are deeply equal to each other and false otherwise.
static boolean	**equals**(Object a, Object b) Returns true if the arguments are equal to each other and false otherwise.
static int	**hash**(Object... values) Generates a hash code for a sequence of input values.
static int	**hashCode**(Object o) Returns the hash code of a non-null argument and 0 for a null argument.
static <T> T	**requireNonNull**(T obj) Checks that the specified object reference is not null.
static <T> T	**requireNonNull**(T obj, String message) Checks that the specified object reference is not null and throws a customized NullPointerException if it is.
static String	**toString**(Object o) Returns the result of calling toString for a non-null argument and "null" for a null argument.
static String	**toString**(Object o, String nullDefault) Returns the result of calling toString on the first argument if the first argument is not null and returns the second argument otherwise.

API de l'interface java.lang.Iterable<T>

Java.util.Iterator<T>	**iterator**() Returns an iterator over a set of elements of type T.