IN413 (Java) : Contrôle final

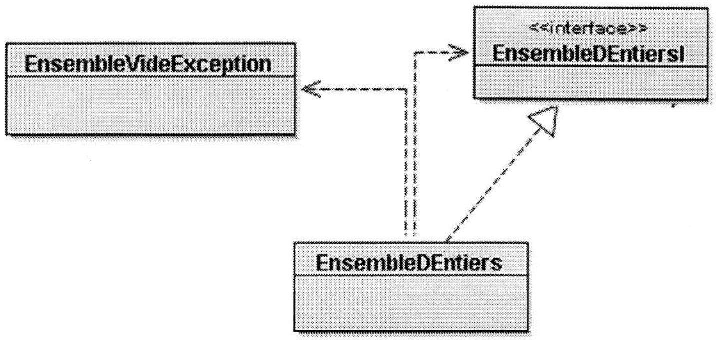
Groupe ESIEE, Jean-Michel DOUIN & Denis BUREAU, Octobre 2006

2 heures, SANS DOCUMENTS, SANS CALCULATRICE

Les documentations Sun de toutes les méthodes et classes utiles sont fournies à la fin de cet énoncé.

L'objectif des classes développées au cours des deux premières questions est de créer une classe « ensemble d'entiers » et de pouvoir vérifier que ses méthodes fonctionnent correctement.

Question 1



(i) Dans le package `question1`, on donne la classe `EnsembleVideException` :

```
public class EnsembleVideException extends Exception {}
```

et l'interface `EnsembleDEntiersI` que devront respecter toutes les classes qui voudront implémenter un « ensemble d'entiers » :

```
import java.util.Iterator;
public interface EnsembleDEntiersI extends Iterable<Integer> {
  public boolean ajoute( Integer e );       // ajout d'un élément ***
  public boolean retire( Integer e );       // retrait d'un élément ***
  public boolean contient( Integer e );     // test de présence d'un élément
  public boolean estSousEnsemble( EnsembleDEntiersI en );
                                            // en inclus dans this
  public int cardinal();                    // nombre d'éléments
  public Integer choisir() throws EnsembleVideException; // *
  public Iterator<Integer> iterator();
  public boolean invariant() throws RuntimeException;    // **
}
```

* doit retourner un élément choisi <u>au hasard</u> dans l'ensemble (voir classe `Random`)

** L'invariant de représentation est une propriété qui doit être vraie au début et à la fin de chaque méthode, et en sortie du constructeur.

*** retourne `true` si et seulement si l'ensemble est modifié

(ii) On désire maintenant créer une classe concrète `EnsembleDEntiers` qui utilise un vecteur (`java.util.Vector<T>`) pour stocker ses éléments de type `Integer`.

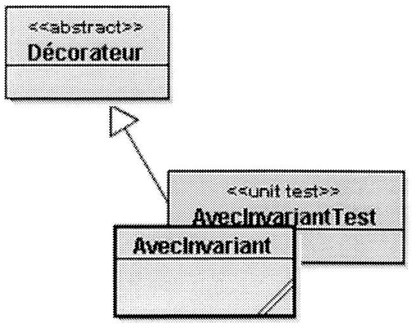
(/ 7) Écrire le fichier `EnsembleDEntiers.java` (**sauf la méthode `invariant()`** qui est spécifiée dans la question suivante).

(iii) **(/ 2)** Écrire la méthode `invariant()` qui devra vérifier :

- que le vecteur a bien été alloué
- que chaque élément n'est présent qu'une seule fois (pas de doublon)

Cette méthode retourne `true` , ou lève une exception `RuntimeException` en cas d'échec.

Question 2



On désire maintenant vérifier l'invariant de représentation dans toutes les méthodes de la classe `EnsembleDEntiers`. Si on insère tous ces tests dans la classe `EnsembleDEntiers`, cette classe deviendra très peu performante.

Pour éviter cet inconvénient, le pattern *Décorateur* conseille d'envelopper notre classe par une classe `Décorateur` qui contiendra un `EnsembleDEntiers` et lui déléguera toutes les opérations, et une sous-classe `AvecInvariant` qui ajoutera le test des invariants pour chaque méthode.

(i) **(/ 2)** Comme nous envisageons de « décorer » notre classe de plusieurs manières, nous commençons par créer une classe abstraite `Décorateur` dont `AvecInvariant` hérite.
Écrivez d'abord la classe `Décorateur`, mais pour éviter des répétitions fastidieuses, il vous est demandé d'écrire le début du fichier `Décorateur.java` **(attributs et constructeur inclus) puis uniquement les méthodes `ajoute` et `invariant`.**

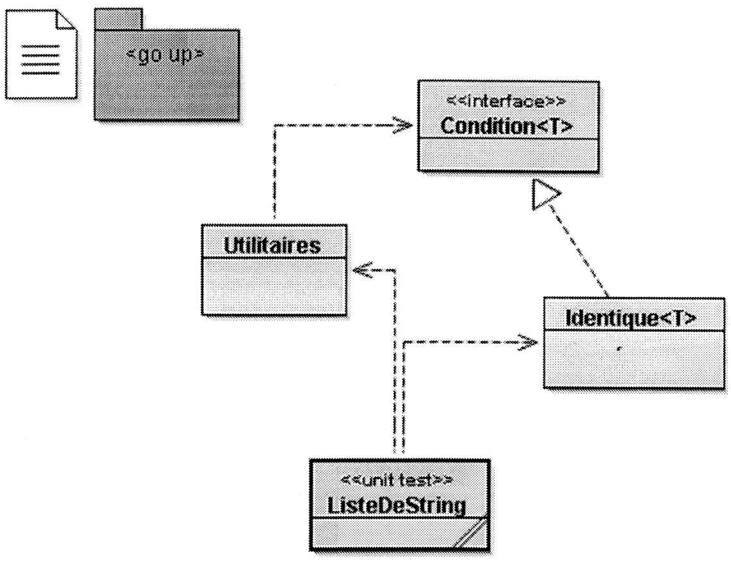
(ii) **(/ 2)** Il faut maintenant écrire la classe `AvecInvariant` en ajoutant dans chaque méthode les vérifications de l'invariant. Pour éviter des répétitions fastidieuses, il vous est demandé d'écrire le début du fichier `AvecInvariant.java` **(attributs et constructeur inclus) puis uniquement les méthodes `ajoute` et `invariant`.**

Un exemple de classe de test :

```
public class AvecInvariantTest extends junit.framework.TestCase {
  public void testSommaire() throws Exception {
    EnsembleDEntiersI avecInval = new AvecInvariant( new EnsembleDEntiers() );
    assertTrue( avecInval.ajoute(3) );
    assertFalse( avecInval.ajoute(3) );
    assertEquals( 1, avecInval.cardinal() );
    assertTrue( avecInval.ajoute(2) );
    assertEquals( 2, avecInval.cardinal() );
    assertTrue( avecInval.retire(3) );
    assertEquals( 1, avecInval.cardinal() );
    assertFalse( avecInval.retire(3) );
```

Question 3

La classe `Utilitaires` ci-dessous contient une méthode de « filtrage », permettant de retirer tous les éléments d'une collection qui satisfont une condition.



(i) La classe Utilitaires (à compléter) : (/ 2)

```
package question3;
import java.util.Collection;
import java.util.Iterator;

public class Utilitaires {
  /** retire de collection tous les éléments vérifiant condition */
  public static <T> void filtrer( Collection<T> collection,
                                   Condition<T> condition ) {
    // à compléter
} }
```

l'interface Condition :

```
public interface Condition<T> {
  public boolean estVraie( T t );
}
```

.../...

et enfin la classe de test :

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

public class ListeDeString extends junit.framework.TestCase {

  public void test1() {
    List<String> l = new ArrayList<String>();
    l.add( "abc" ); l.add( "aac" ); l.add( "bbc" );
    l.add( "aac" ); l.add( "abc" );
    assertEquals( l.toString(), "[abc, aac, bbc, aac, abc]" );

    Utilitaires.filtrer( l, new Identique<String>( "aac" ) );
    assertEquals( l.toString(), "[abc, bbc, abc]" );
  }

  public void test2(){
    List<String> l1 = new ArrayList<String>();
    List<String> l2 = new ArrayList<String>();
    List<String> l3 = new ArrayList<String>();
    l1.add( "abc" );
    l2.add( "aac" ); l2.add( "bbc" );
    l3.add( "abc" );
    List< List<String> > l = new ArrayList< List<String> >();
    l.add( l1 ); l.add( l2 ); l.add( l3 );
    assertEquals( l.toString(), "[[abc], [aac, bbc], [abc]]" );

    Utilitaires.filtrer( l, new Identique< List<String> >( l3 ) );
    assertEquals( l.toString(), "[[aac, bbc]]" );
  }
}
```

(ii) Écrivez la classe Identique<T> (/ 2)

Une instance de celle-ci, transmise à la méthode filtrer, permet de retirer toute occurrence d'un élément dans une collection, comme le suggère la classe de test ci-dessus.

Question 4 (/ 2)

(i) Quels sont les points communs entre interface et classe abstraite ?

(ii) Quelles sont les différences entre interface et classe abstraite ?

Pensez notamment aux attributs, aux constantes, aux constructeurs, aux méthodes abstraites/concrètes, au nombre d'interfaces/classes abstraites auxquelles une classe peut faire référence, au mot-clé à employer pour cela, aux redéfinitions obligatoires ou non, à `instanceof`, au transtypage, etc…

Qualité générale de la programmation (/ 2)

- Très bonne	==>	+2 (bonus !)
- Normale	==>	+1
- Mauvaise	==>	+0

Documentations utiles

|| Iterable || Iterator || RuntimeException || Random || Vector || Integer || Collection || List || ArrayList ||

---

java.lang

Interface Iterable<T>

All Known Subinterfaces:
BeanContext, BeanContextServices, BlockingQueue<E>, Collection<E>, List<E>, Queue<E>, Set<E>, SortedSet<E>

All Known Implementing Classes:
AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentLinkedQueue, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingQueue, LinkedHashSet, LinkedList, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector

Implementing this interface allows an object to be the target of the "foreach" statement.

---

iterator

`Iterator<T> iterator()`

Returns an `iterator` over a set of elements of type `T`.

Returns:
`an Iterator.`

---

java.util

Interface Iterator<E>

All Known Subinterfaces:
`ListIterator<E>`

An iterator over a collection. Iterator takes the place of Enumeration in the Java collections framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

---

hasNext

`boolean hasNext()`

Returns true if the iteration has more elements. (In other words, returns true if next would return an element rather than throwing an exception.)

Returns:
true if the iterator has more elements.

---

next

`E next()`

Returns the next element in the iteration. Calling this method repeatedly until the hasNext() method returns false will return each element in the underlying collection exactly once.

Returns:
the next element in the iteration.

Throws:
NoSuchElementException - iteration has no more elements.

---

remove

```
void remove()
```

Removes from the underlying collection the last element returned by the iterator (optional operation). This method can be called only once per call to next. The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

Throws:

UnsupportedOperationException - if the remove operation is not supported by this Iterator.

IllegalStateException - if the next method has not yet been called, or the remove method has already been called after the last call to the next method.

---

java.lang

Class RuntimeException

```
extends Exception
```

RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.

A method is not required to declare in its throws clause any subclasses of RuntimeException that might be thrown during the execution of the method but not caught.

RuntimeException

```
public RuntimeException(String message)
```

Constructs a new runtime exception with the specified detail message. The cause is not initialized, and may subsequently be initialized by a call to Throwable.initCause(java.lang.Throwable).

Parameters:

message - the detail message. The detail message is saved for later retrieval by the Throwable.getMessage() method.

---

java.util

Class Random

An instance of this class is used to generate a stream of pseudorandom numbers. The class uses a 48-bit seed, which is modified using a linear congruential formula. (See Donald Knuth, The Art of Computer Programming, Volume 2, Section 3.2.1.)

If two instances of Random are created with the same seed, and the same sequence of method calls is made for each, they will generate and return identical sequences of numbers. In order to guarantee this property, particular algorithms are specified for the class Random. Java implementations must use all the algorithms shown here for the class Random, for the sake of absolute portability of Java code. However, subclasses of class Random are permitted to use other algorithms, so long as they adhere to the general contracts for all the methods.

The algorithms implemented by class Random use a protected utility method that on each invocation can supply up to 32 pseudorandomly generated bits.

Random

```
public Random()
```

Creates a new random number generator. This constructor sets the seed of the random number generator to a value very likely to be distinct from any other invocation of this constructor.

T get

`public Object get(int index)`

Returns the element at the specified position in this Vector.

Parameters:

`index` - index of element to return.

Throws:

`ArrayIndexOutOfBoundsException` - index is out of range (index < 0 || index >= size()).

---

add

`public boolean add( T o)`

Appends the specified element to the end of this Vector.

Parameters:

`o` - element to be appended to this Vector.

Returns:

true (as per the general contract of Collection.add).

---

remove

`public boolean remove( T o)`

Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged. More formally, removes the element with the lowest index i such that `(o==null ? get(i)==null : o.equals(get(i)))` (if such an element exists).

Parameters:

`o` - element to be removed from this Vector, if present.

Returns:

true if the Vector contained the specified element.

---

iterator (`inherited from class java.util.AbstractList`)

`public Iterator<E> iterator()`

Returns an iterator over the elements in this list in proper sequence.

This implementation returns a straightforward implementation of the iterator interface, relying on the backing list's size(), get(int), and remove(int) methods.

Note that the iterator returned by this method will throw an UnsupportedOperationException in response to its remove method unless the list's remove(int) method is overridden.

This implementation can be made to throw runtime exceptions in the face of concurrent modification, as described in the specification for the (protected) modCount field.

Specified by:

iterator in interface Iterable<E>, iterator in interface Collection<E>, iterator in interface List<E>, iterator in class AbstractCollection<E>

Returns:

an iterator over the elements in this list in proper sequence.

---

nextInt

`public int nextInt(int n)`

Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence. The general contract of nextInt is that one int value in the specified range is pseudorandomly generated and returned. All n possible int values are produced with (approximately) equal probability.

Parameters:
n - the bound on the random number to be returned. Must be positive.

Returns:
a pseudorandom, uniformly distributed int value between 0 (inclusive) and n (exclusive).

Throws:
IllegalArgumentException - n is not positive.

---

java.util

Class Vector<E>

```
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

The Vector class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.

Each vector tries to optimize storage management by maintaining a capacity and a capacityIncrement. The capacity is always at least as large as the vector size; it is usually larger because as components are added to the vector, the vector's storage increases in chunks the size of capacityIncrement. An application can increase the capacity of a vector before inserting a large number of components; this reduces the amount of incremental reallocation.

Vector

`public Vector()`

Constructs an empty vector so that its internal data array has size `10` and its standard capacity increment is zero.

---

size

`public int size()`

Returns the number of components in this vector.

Returns:
the number of components in this vector.

---

contains

`public boolean contains(Object elem)`

Tests if the specified object is a component in this vector.

Parameters:

`elem` - an object.

Returns:
`true` if and only if the specified object is the same as a component in this vector, as determined by the `equals` method; `false` otherwise.

---

java.lang

Class Integer

`extends Number`
`implements Comparable<Integer>`

The Integer class wraps a value of the primitive type int in an object. An object of type Integer contains a single field whose type is int.

In addition, this class provides several methods for converting an int to a String and a String to an int, as well as other constants and methods useful when dealing with an int.

Integer

`public **Integer**(int value)`

Constructs a newly allocated `Integer` object that represents the primitive `int` argument.

Parameters:

`value` - the value to be represented by the `Integer`.

---

intValue

`public int **intValue**()`

Returns the value of this Integer as an int.

Returns:

the `int` value represented by this object.

---

equals

`public boolean **equals**(Object obj)`

Compares this object to the specified object. The result is true if and only if the argument is not null and is an Integer object that contains the same int value as this object.

Overrides:

equals in class Object

Parameters:

obj - the object to compare with.

Returns:

true if the objects are the same; false otherwise.

---

java.util

Interface Collection<E>

`extends Iterable<E>`

The root interface in the collection hierarchy. A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any direct implementations of this interface: it provides implementations of more specific subinterfaces like Set and List. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

---

iterator

`Iterator<E> **iterator**()`

Returns an iterator over the elements in this collection. There are no guarantees concerning the order in which the elements are returned (unless this collection is an instance of some class that provides a guarantee).

Specified by:

iterator in interface Iterable<E>

Returns:

an Iterator over the elements in this collection

---

java.util

Interface List<E>

`extends Collection<E>`

An ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements e1 and e2 such that e1.equals(e2), and they typically allow multiple null elements if they allow null elements at all. It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare.

The List interface places additional stipulations, beyond those specified in the Collection interface, on the contracts of the iterator, add, remove, equals, and hashCode methods. Declarations for other inherited methods are also included here for convenience.

---

java.util

Class ArrayList<E>

```
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

---

add

```
public boolean add(E o)
```

Appends the specified element to the end of this list.

Specified by:

add in interface Collection<E>, add in interface List<E>

Overrides:

add in class AbstractList<E>

Parameters:

o - element to be appended to this list.

Returns:

true (as per the general contract of Collection.add).
