```
# Executables
OSTYPE = $(shell uname -s)
JAVAC = javac
JAVA = java
# A2PS = a2ps-utf8
A2PS = a2ps
GHOSTVIEW = gv
DOCP = javadoc
#ARCH = zip
ARCH = tar zcvf
PS2PDF = ps2pdf -dPDFX=true -sPAPERSIZE=a4
DATE = $(shell date +%Y-%m-%d)
# Options de compilation
#CFLAGS = -verbose
CFLAGS =
ifeq ($(findstring Darwin,$(OSTYPE)),Darwin)
    # MacOS systems
    CLASSPATH=.:/opt/local/share/java/junit.jar:/opt/local/share/java/hamcrest-core.jar
else
    # Other systems
    CLASSPATH=.
endif

JAVAOPTIONS = --verbose

PROJECT=Ensembles
# nom du fichier d'impression
OUTPUT = $(PROJECT)
# nom du rÃ©pertoire ou se situera la documentation
DOC = doc
# lien vers la doc en ligne du JDK
WEBLINK = "http://docs.oracle.com/javase/6/docs/api/"
# lien vers la doc locale du JDK
LOCALLINK = "file:///Users/davidroussel/Documents/docs/java/api/"
# nom de l'archive
ARCHIVE = $(PROJECT)
# format de l'archive pour la sauvegarde
#ARCHFMT = zip
ARCHFMT = tgz
# RÃ©pertoire source
SRC = src
# RÃ©pertoire bin
BIN = bin
# RÃ©pertoire Listings
LISTDIR = listings
# RÃ©pertoire Archives
ARCHDIR = archives
# RÃ©pertoire Figures
FIGDIR = graphics
# noms des fichiers sources
MAIN = RunAllTests
SOURCES = $(foreach name, $(MAIN), $(SRC)/$(name).java) \
$(SRC)/listes/package-info.java \
$(SRC)/listes/IListe.java \
$(SRC)/listes/Liste.java \
$(SRC)/tableaux/package-info.java \
$(SRC)/tableaux/Tableau.java \
$(SRC)/ensembles/package-info.java \
$(SRC)/ensembles/Ensemble.java \
$(SRC)/ensembles/EnsembleGenerique.java \
$(SRC)/ensembles/EnsembleVector.java \
$(SRC)/ensembles/EnsembleListe.java \
$(SRC)/ensembles/EnsembleTableau.java \
$(SRC)/ensembles/EnsembleFactory.java \
$(SRC)/ensembles/EnsembleTri.java \
$(SRC)/ensembles/EnsembleTriVector.java \
$(SRC)/ensembles/EnsembleTriListe.java \
$(SRC)/ensembles/EnsembleTriTableau.java \
$(SRC)/ensembles/EnsembleTriGenerique.java \
$(SRC)/ensembles/EnsembleTriVector2.java \
$(SRC)/ensembles/EnsembleTriListe2.java \
$(SRC)/ensembles/EnsembleTriTableau2.java \
$(SRC)/ensembles/EnsembleTriFactory.java \
$(SRC)/tests/package-info.java \
$(SRC)/tests/AllTests.java \
$(SRC)/tests/AllEnsembleTest.java \
$(SRC)/tests/ListeTest.java \
$(SRC)/tests/TableauTest.java \
$(SRC)/tests/EnsembleTriTest.java

OTHER = Sujet.pdf

.PHONY : doc ps

# Les targets de compilation
# pour gÃ©nÃ©rer l'application
all : $(foreach name, $(MAIN), $(BIN)/$(name).class)

#rÃ¨gle de compilation gÃ©nÃ©rique
$(BIN)/%.class : $(SRC)/%.java
```



```
    $(JAVAC) -sourcepath $(SRC) -classpath $(BIN):$(CLASSPATH) -d $(BIN) $(CFLAGS) $<

# Edition des sources $(EDITOR) doit Ãªtre une variable d'environnement
edit :
    $(EDITOR) $(SOURCES) Makefile &

# nettoyer le rÃ©pertoire
clean :
    find bin/ -type f -name "*.class" -exec rm -f {} \;
    rm -rf *~ $(DOC)/* $(LISTDIR)/*

realclean : clean
    rm -f $(ARCHDIR)/*.$(ARCHFMT)

# gÃ©nÃ©rer le listing
$(LISTDIR) :
    mkdir $(LISTDIR)

ps : $(LISTDIR)
    $(A2PS) -2 --file-align=fill --line-numbers=1 --font-size=10 \
    --chars-per-line=100 --tabsize=4 --pretty-print \
    --highlight-level=heavy --prologue="gray" \
    -o$(LISTDIR)/$(OUTPUT).ps Makefile $(SOURCES)

pdf : ps
    $(PS2PDF) $(LISTDIR)/$(OUTPUT).ps $(LISTDIR)/$(OUTPUT).pdf

# gÃ©nÃ©rer le listing lisible pour GÃ©rard
bigps :
    $(A2PS) -1 --file-align=fill --line-numbers=1 --font-size=10 \
    --chars-per-line=100 --tabsize=4 --pretty-print \
    --highlight-level=heavy --prologue="gray" \
    -o$(LISTDIR)/$(OUTPUT).ps Makefile $(SOURCES)

bigpdf : bigps
    $(PS2PDF) $(LISTDIR)/$(OUTPUT).ps $(LISTDIR)/$(OUTPUT).pdf

# voir le listing
preview : ps
    $(GHOSTVIEW) $(LISTDIR)/$(OUTPUT); rm -f $(LISTDIR)/$(OUTPUT) $(LISTDIR)/$(OUTPUT)~

# gÃ©nÃ©rer la doc avec javadoc
doc : $(SOURCES)
    $(DOCP) -private -d $(DOC) -author -link $(LOCALLINK) $(SOURCES)
#   $(DOCP) -private -d $(DOC) -author -linkoffline $(WEBLINK) $(LOCALLINK) $(SOURCES)

# gÃ©nÃ©rer une archive de sauvegarde
$(ARCHDIR) :
    mkdir $(ARCHDIR)

archive : pdf $(ARCHDIR)
    $(ARCH) $(ARCHDIR)/$(ARCHIVE)-$(DATE).$(ARCHFMT) $(SOURCES) $(LISTDIR)/*.pdf $(OTHER) $(BIN) Mak
efile

# exÃ©cution des programmes de test
run : all
    $(foreach name, $(MAIN), $(JAVA) -classpath $(BIN):$(CLASSPATH) $(name) $(JAVAOPTIONS) )
```

30 sep 15 16:46 **RunAllTests.java** Page 1/1

```java
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

import tests.AllTests;

/**
 * ExÃ©cution de tous les tests du package "tests"
 * @author davidroussel
 */
public class RunAllTests
{
    /**
     * Programme principal de lancement des tests
     * @param args non utilisÃ©s
     */
    public static void main(String[] args)
    {
        System.out.println("Test des ensembles");

        Result result = JUnitCore.runClasses(AllTests.class);

        int failureCount = result.getFailureCount();

        if (failureCount == 0)
        {
            System.out.println("Every thing went fine");
        }
        else
        {
            for (Failure failure : result.getFailures())
            {
                System.err.println(failure);
            }
        }
    }
}
```

20 oct 14 17:22 **package-info.java** Page 1/1

```java
/**
 * Package contenant l'implÃ©mentation des listes simplement chainÃ©es dÃ©finies
 * dans l'interface {@link listes.IListe} et implÃ©mentÃ©es dans la classe
 * {@link listes.Liste}
 */
package listes;
```

```java
package listes;

import java.util.Iterator;

/**
 * Interface d'une liste gÃ©nÃ©rique d'Ã©lÃ©ments.
 *
 * @note On considÃ¨rera que la liste ne peut pas contenir d'elt null
 * @author David Roussel
 * @param <E> le type des Ã©lÃ©ments de la liste.
 */
public interface IListe<E> extends Iterable<E>
{
    /**
     * Ajout d'un Ã©lÃ©ment en fin de liste
     *
     * @param elt l'Ã©lÃ©ment Ã  ajouter en fin de liste
     * @throws NullPointerException si l'on tente d'ajouter un Ã©lÃ©ment null
     */
    public abstract void add(E elt) throws NullPointerException;

    /**
     * Insertion d'un Ã©lÃ©ment en tÃªte de liste
     *
     * @param elt l'Ã©lÃ©ment Ã  ajouter en tÃªte de liste
     * @throws NullPointerException si l'on tente d'insÃ©rer un Ã©lement null
     */
    public abstract void insert(E elt) throws NullPointerException;

    /**
     * Insertion d'un Ã©lÃ©ment Ã  la (index+1)iÃ¨me place
     *
     * @param elt l'Ã©lÃ©ment Ã  insÃ©rer
     * @param index l'index de l'Ã©lÃ©ment Ã  insÃ©rer
     * @return true si l'Ã©lÃ©ment a pu Ãªtre insÃ©rÃ© Ã  l'index voulu, false sinon
     *         ou si l'Ã©lÃ©ment Ã  insÃ©rer Ã©tait null
     */
    public abstract boolean insert(E elt, int index);

    /**
     * Suppression de la premiÃ¨re occurrence de l'Ã©lÃ©mÃ©nt e
     * (en utilisant l'itÃ©rateur)
     *
     * @param elt l'Ã©lement Ã  rechercher et Ã  supprimer.
     * @return true si l'Ã©lÃ©ment a Ã©tÃ© trouvÃ© et supprimÃ© de la liste
     * @note doit fonctionner mÃªme si e est null
     */
    public default boolean remove(E elt)
    {
        /*
         * TODO ComplÃ©ter ...
         */
        return false;
    }

    /**
     * Suppression de toutes les instances de e dans la liste
     * (en utilisant l'itÃ©rateur)
     *
     * @param elt l'Ã©lÃ©ment Ã  supprimer
     * @return true si au moins un Ã©lÃ©ment a Ã©tÃ© supprimÃ©
     * @note doit fonctionner mÃªme si e est null
     */
    public default boolean removeAll(E elt)
    {
        boolean result = false;
        /*
         * TODO ComplÃ©ter ...
         */
        return result;
    }

    /**
     * Nombre d'Ã©lÃ©ments dans la liste
     * (en utilisant l'itÃ©rateur)
     *
     * @return le nombre d'Ã©lÃ©ments actuellement dans la liste
     */
    public default int size()
    {
        int count = 0;
        /*
         * TODO ComplÃ©ter ...
         */
        return count;
    }

    /**
     * Effacement de la liste;
     * (en utilisant l'itÃ©rateur)
```



```java
     */
    public default void clear()
    {
        /*
         * TODO ComplÃ©ter ...
         */
    }

    /**
     * Test de liste vide
     *
     * @return true si la liste est vide, false sinon
     */
    public default boolean empty()
    {
        /*
         * TODO Remplacer par l'implÃ©mentation ...
         */
        return false;
    }

    /**
     * Test d'Ã©galitÃ© au sens du contenu de la liste
     *
     * @param o la liste dont on doit tester le contenu
     * @return true si o est une liste, que tous les maillons des deux listes
     *         sont identiques (au sens du equals de chacun des maillons), dans
     *         le mÃªme ordre, et que les deux listes ont la mÃªme longueur. false
     *         sinon
     * @note On serait tentÃ© d'en faire une "default method" dans la mesure oÃ¹
     *       l'on peut n'utiliser que l'itÃ©rateur pour parcourir les Ã©lÃ©ments de
     *       la liste MAIS les mÃ©thodes par dÃ©faut n'ont pas le droit de
     *       surcharger les mÃ©thodes de la superclasse Object.
     */
    @Override
    public abstract boolean equals(Object o);

    /**
     * HashCode d'une liste
     *
     * @return le hashCode de la liste
     * @note On serait tentÃ© d'en faire une "default method" dans la mesure oÃ¹
     *       l'on peut n'utiliser que l'itÃ©rateur pour parcourir les Ã©lÃ©ments de
     *       la liste MAIS les mÃ©thodes par dÃ©faut n'ont pas le droit de
     *       surcharger les mÃ©thodes de la superclasse Object.
     */
    @Override
    public abstract int hashCode();

    /**
     * ReprÃ©sentation de la chaine sous forme de chaine de caractÃ¨re.
     *
     * @return une chaine de caractÃ¨re reprÃ©sentant la liste chainÃ©e
     * @note On serait tentÃ© d'en faire une "default method" dans la mesure oÃ¹
     *       l'on peut n'utiliser que l'itÃ©rateur pour parcourir les Ã©lÃ©ments de
     *       la liste MAIS les mÃ©thodes par dÃ©faut n'ont pas le droit de
     *       surcharger les mÃ©thodes de la superclasse Object.
     */
    @Override
    public abstract String toString();

    /**
     * Obtention d'un itÃ©rateur pour parcourir la liste : <code>
     * Liste<Type> l = new Liste<Type>();
     * ...
     * for (Iterator<Type> it = l.iterator(); it.hasNext(); )
     * {
     *       ... it.next() ...
     * }
     * ou bien
     * for (Type elt : l)
     * {
     *       ... elt ...
     * }
     * </code>
     *
     * @return un nouvel itÃ©rateur sur la liste
     * @see {@link Iterable#iterator()}
     */
    @Override
    public abstract Iterator<E> iterator();
}
```

```java
/**
 * Package contenant la classe {@link tableaux.Tableau} : tableau de donnÃ©es de
 * taille variable
 */
package tableaux;
```



```java
package tableaux;

import java.util.Collection;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * Tableau de donnÃ©es
 *
 * @author davidroussel
 * @param <E> le type des donnÃ©es stockÃ©es dans le tableau
 */
public class Tableau<E> implements Iterable<E>
{
    /**
     * Le tableau de donnÃ©es
     */
    protected E[] table;

    /**
     * nombre d'Ã©lÃ©ments actuellement dans le tableau. Et index du prochain
     * Ã©lÃ©ment Ã  insÃ©rer
     */
    protected int size;

    /**
     * Nombre de cases max du tableau
     */
    protected int capacity;

    /**
     * Nombres de cases initiales par dÃ©faut du tableau de donnÃ©es. Et nombre de
     * cases Ã  rajouter en cas de manque de cases
     */
    public static final int INCREMENT = 5;

    /**
     * constructeur par dÃ©faut d'un tableau de donnÃ©es
     */
    @SuppressWarnings("unchecked")
    public Tableau()
    {
        table = (E[]) new Object[INCREMENT];
        size = 0;
    }

    /**
     * constructeur de copie Ã  partir d'un autre {@link Iterable}
     *
     * @param elements l'itÃ©rable dont on doit copier les Ã©lÃ©ments
     */
    public Tableau(Iterable<E> elements)
    {
        this();
        for (E elt : elements)
        {
            ajouter(elt);
        }
    }

    /**
     * Nombre d'Ã©lÃ©ments actuellement dans le tableau
     *
     * @return Le nombre d'Ã©lÃ©ments actuellement dans le tableau
     */
    public int taille()
    {
        return size;
    }

    /**
     * Nombre d'Ã©lÃ©ments maximum (actuellement) dans le tableau
     *
     * @return le nombre de l'Ã©lÃ©ments amx dans le tableau actuellement
     */
    public int capacite()
    {
        return capacity;
    }

    /**
     * Ajout d'un element Ã  la fin du tableau
     *
     * @param element l'Ã©lÃ©ment Ã  insÃ©rer
     */
    public void ajouter(E element)
    {
        if (size ≥ capacity)
        {
            // ajouterCapacite(Math.max(INCREMENT, (size - capacity) + 1));
```

```
91              int scl = (size - capacity) + 1;
92              ajouterCapacite((INCREMENT ≥ scl ? INCREMENT : scl));
93          }
94          table[size] = element;
95          size++;
96      }
97
98      /**
99       * Ajout de nbCases au tableau
100      *
101      * @param nbCases nombre de cases Ã  ajouter.
102      */
103     protected void ajouterCapacite(int nbCases)
104     {
105         if (nbCases > 0)
106         {
107             capacity += nbCases;
108             @SuppressWarnings("unchecked")
109             E[] newTable = (E[]) new Object[capacity];
110             for (int i = 0; i < size; i++)
111             {
112                 newTable[i] = table[i];
113                 table[i] = null; // avoid weak references
114             }
115             table = newTable;
116         }
117     }
118
119     /**
120      * Retrait de la prmiÃ¨re occurrence d'un Ã©lÃ©ment
121      *
122      * @param element l'Ã©lÃ©ment Ã  retirer du tableau
123      * @return true si l'Ã©lÃ©ment a Ã©tÃ© trouvÃ© et retirÃ©
124      */
125     public boolean retrait(E element)
126     {
127         for (Iterator<E> it = iterator(); it.hasNext();)
128         {
129             if (it.next().equals(element))
130             {
131                 it.remove();
132                 return true;
133             }
134         }
135
136         return false;
137     }
138
139     /**
140      * Effacement de tous les Ã©lÃ©ments tu tableau
141      */
142     public void efface()
143     {
144         for (Iterator<E> it = iterator(); it.hasNext();)
145         {
146             it.next();
147             it.remove();
148         }
149     }
150
151     /**
152      * Insertion d'un Ã©lÃ©ment en dÃ©but de tableau
153      *
154      * @param element l'Ã©lÃ©ment Ã  insÃ©rer
155      */
156     public void insertElement(E element)
157     {
158         try
159         {
160             insertElement(element, 0);
161         }
162         catch (IndexOutOfBoundsException ioobe)
163         {
164             System.err.println("Tableau::insertElement : " + ioobe);
165         }
166     }
167
168     /**
169      * Insertion d'un Ã©lÃ©ment Ã  la place index
170      *
171      * @param element l'Ã©lÃ©ment Ã  insÃ©rer dans le tableau
172      * @param index l'index oÃ¹ insÃ©rer l'Ã©lement
173      * @throws IndexOutOfBoundsException si l'index oÃ¹ insÃ©rer l'Ã©lÃ©ment est
174      *             invalide
175      */
176     public void insertElement(E element, int index)
177             throws IndexOutOfBoundsException
178     {
179         if ((index ≤ size) ∧ (index ≥ 0))
180         {
```



```
181             if (index ≡ size)
182             {
183                 ajouter(element);
184             }
185             else // index >=0 & < size
186             {
187                 if ((size + 1) ≥ capacity)
188                 {
189                     ajouterCapacite(INCREMENT);
190                 }
191                 // dÃ©calage des elements
192                 for (int i = size; i > index; i--)
193                 {
194                     table[i] = table[i - 1];
195                 }
196
197                 table[index] = element;
198                 size++;
199             }
200         }
201         else
202         {
203             throw new IndexOutOfBoundsException("Invalid Index :"
204                     + Integer.toString(index));
205         }
206
207     }
208
209     /**
210      * Factory method fournissant un itÃ©rateur sur le tableau
211      *
212      * @return un nouvel itÃ©rateur sur le tableau
213      */
214     @Override
215     public Iterator<E> iterator()
216     {
217         return new TabIterator<E>();
218     }
219
220     /**
221      * Test d'Ã©galitÃ© avec un autre objet.
222      * @return true ssi l'objet est un {@link Tableau} et qu'il contient
223      * les mÃªmes Ã©lÃ©mÃ©nts dans le mÃªme ordre.
224      * @see java.lang.Object#equals(java.lang.Object)
225      */
226     @Override
227     public boolean equals(Object obj)
228     {
229         if (obj ≡ null)
230         {
231             return false;
232         }
233
234         if (obj ≡ this)
235         {
236             return true;
237         }
238
239         if (getClass().isInstance(obj))
240         {
241             Tableau<?> tab = (Tableau<?>) obj;
242
243             Iterator<E> it1 = iterator();
244             Iterator<?> it2 = tab.iterator();
245
246             for (; it1.hasNext() ∧ it2.hasNext();)
247             {
248                 if (¬it1.next().equals(it2.next()))
249                 {
250                     return false;
251                 }
252             }
253
254             return ¬it1.hasNext() ∧ ¬it2.hasNext();
255         }
256         else
257         {
258             return false;
259         }
260     }
261
262     /**
263      * Code de hashage d'un tableau.
264      * Le code he hashage est compatible avec celui fournit par toute {@link Collection}
265      * contenant les mÃªmes Ã©lÃ©ments dans le mÃªme ordre.
266      * @return le code de hashage rÃ©sultants des Ã©lÃ©ments du Tableau
267      * @see java.lang.Object#hashCode()
268      */
269     @Override
270     public int hashCode()
```

```
271     {
272         final int prime = 31;
273         int result = 1;
274         for (E elt : this)
275         {
276             result = (prime * result) + (elt == null ? 0 : elt.hashCode());
277         }
278         return result;
279     }
280 
281     /**
282      * Chaine de caractÃ¨re reprÃ©sentant les Ã©lÃ©ments du tableau ainsi que sa
283      * taille et sa capacitÃ© courante
284      * @return une nouvelle chaine de caractÃ¨re reprÃ©sentant le Tableau
285      * @see java.lang.Object#toString()
286      */
287     @Override
288     public String toString()
289     {
290         StringBuilder sb = new StringBuilder();
291 
292         sb.append("[");
293         for (Iterator<E> it = iterator(); it.hasNext();)
294         {
295             sb.append(it.next().toString());
296             if (it.hasNext())
297             {
298                 sb.append(",");
299             }
300         }
301         sb.append("](");
302         sb.append(Integer.toString(size));
303         sb.append(",");
304         sb.append(Integer.toString(capacity));
305         sb.append(")");
306 
307         return new String(sb);
308     }
309 
310     /**
311      * ItÃ©rateur sur un {@link Tableau}
312      *
313      * @author davidroussel
314      * @param <F> le type des Ã©lÃ©ments Ã  itÃ©rer
315      */
316     private class TabIterator<F> implements Iterator<F>
317     {
318         /**
319          * L'index courant de l'itÃ©rateur. index de l'Ã©lÃ©ment courant dans le
320          * tableau
321          */
322         private int index;
323 
324         /**
325          * Indique si next vient d'Ãªtre appelÃ© ce qui permet (Ã©ventuellement)
326          * d'appeler remove.
327          */
328         private boolean nextCalled;
329 
330         /**
331          * Constructeur par dÃ©faut d'un itÃ©rateur sur un tableau
332          */
333         public TabIterator()
334         {
335             index = 0;
336             nextCalled = false;
337         }
338 
339         /**
340          * Clause de continuation
341          *
342          * @return true si l'itÃ©rateur peut encore itÃ©rer (utiliser la mÃ©thode
343          *         {@link #next()})
344          */
345         @Override
346         public boolean hasNext()
347         {
348             return index < size;
349         }
350 
351         /**
352          * IncrÃ©mentation de l'itÃ©rateur
353          * @return la donnÃ©e correpondant Ã  la position courante de l'itÃ©rateur
354          * @throws NoSuchElementException si l'itÃ©rateur ne peut plus itÃ©rer,
355          * lorsque celui ci a dÃ©jÃ  atteint le dernier Ã©lÃ©ment Ã  itÃ©rer
356          */
357         @Override
358         public F next() throws NoSuchElementException
359         {
360             if (hasNext())
```



```
361             {
362                 @SuppressWarnings("unchecked")
363                 F element = (F) table[index];
364                 index++;
365                 nextCalled = true;
366                 return element;
367             }
368             else
369             {
370                 throw new NoSuchElementException();
371             }
372         }
373 
374         /**
375          * Suppression du dernier Ã©lÃ©ment renvoyÃ© par {@link #next()}.
376          * Attention, remove ne peut Ãªtre appelÃ© qu'aprÃ¨s avoir appelÃ©
377          * {@link #next()}.
378          *
379          * @post l'Ã©lÃ©ment prÃ©cÃ©dent l'Ã©lÃ©ment courant de l'itÃ©rateur a Ã©tÃ©
380          *       supprimÃ©.
381          */
382         @Override
383         public void remove() throws IllegalStateException
384         {
385             if (nextCalled) // index >= 1
386             {
387                 for (int i = index - 1; i < (size - 1); i++)
388                 {
389                     table[i] = table[i + 1];
390                 }
391                 size--;
392                 index--;
393                 nextCalled = false;
394             }
395             else
396             {
397                 throw new IllegalStateException("Next not called yet");
398             }
399         }
400     }
401 }
```

20 oct 14 17:21 **package−info.java** Page 1/1

```java
/**
 * Package contenant la dÃ©finition d'un {@link ensembles.Ensemble} comme Ã©tant
 * une collection (a priori non ordonnÃ©e, mÃªme si le conteneur sous jacent peut
 * Ãªtre ordonnÃ©). {@link ensembles.EnsembleGenerique} fournit une implÃ©mentation
 * partielle des ensembles sans connaÃ®tre encore le conteneur sous-jacent (qui
 * peut Ãªtre un {@link java.util.Vector}, ou bien une {@link listes.Liste}, ou
 * encore un {@link tableaux.Tableau}. {@link ensembles.EnsembleGenerique}
 * n'implÃ©mente pas les opÃ©rations :
 * <ul>
 * <li>d'ajout {@link ensembles.EnsembleGenerique#ajout(Object)} puisqu'elle est
 * spÃ©cifique au conteneur sous-jacent</li>
 * <li>de construction d'un itÃ©rateur
 * {@link ensembles.EnsembleGenerique#iterator()} puisqu'elle est aussi
 * spÃ©cifique au conteneur sous-jacent</li>
 * <li>les opÃ©rations ensembliste comme
 * {@link ensembles.Ensemble#union(Ensemble)},
 * {@link ensembles.Ensemble#intersection(Ensemble)},
 * {@link ensembles.Ensemble#complement(Ensemble)} et
 * {@link ensembles.Ensemble#difference(Ensemble)} de part le fait qu'elle est
 * une classe abstraite et ne peut donc pas "crÃ©er" l'ensemble rÃ©sultat de
 * l'opÃ©ration ensembliste. En revanche elle propose une implÃ©mentation basÃ©e
 * sur les mÃ©thodes de classes dans lesquelle l'ensemble rÃ©sultat est dÃ©jÃ  crÃ©Ã©
 * (par une des classes filles)</li>
 * </ul>
 * {@link ensembles.EnsembleGenerique} implÃ©mente donc
 * <ul>
 * <li>{@link ensembles.Ensemble#union(Ensemble, Ensemble, Ensemble)}</li>
 * <li>{@link ensembles.Ensemble#intersection(Ensemble, Ensemble, Ensemble)}</li>
 * <li>{@link ensembles.Ensemble#complement(Ensemble, Ensemble, Ensemble)}</li>
 * <li>{@link ensembles.Ensemble#difference(Ensemble, Ensemble, Ensemble)}</li>
 * </ul>
 */
package ensembles;
```

04 nov 15 17:54 **Ensemble.java** Page 1/4

```java
package ensembles;

import java.util.Iterator;

/**
 * Interface dÃ©finissant un ensemble comme une collection non triÃ©e d'Ã©lÃ©ments
 * sans doublons. Le fait que les Ã©lÃ©ments sont considÃ©rÃ©s comme non triÃ©s
 * impliquera que la comparaison de deux ensembles ne devra pas prendre en
 * compte l'ordre (apparent) des Ã©lÃ©ments.
 *
 * @author davidroussel
 */
public interface Ensemble<E> extends Iterable<E>
{
    /**
     * Ajout d'un Ã©lÃ©ment Ã  un ensemble ssi celui ci n'est pas null et qu'il
     * n'est pas dÃ©jÃ  prÃ©sent
     *
     * @param element l'Ã©lÃ©ment Ã  ajouter Ã  l'ensemble (on considÃ¨rera que l'on
     *          ne peut pas ajouter d'Ã©lÃ©ment null)
     * @return true si l'Ã©lÃ©ment a pu Ãªtre ajoutÃ© Ã  l'ensemble, false sinon ou
     *         si l'on a tentÃ© d'insÃ©rer un Ã©lÃ©ment null (auquel cas il n'est
     *         pas insÃ©rÃ©)
     */
    public abstract boolean ajout(E element);

    /**
     * Retrait d'un Ã©lÃ©ment de l'ensemble en utilisant le remove de l'itÃ©rateur
     * fournit par {@link #iterator()}
     *
     * @param element l'Ã©lÃ©ment Ã  supprimer de l'ensemble
     * @return true si l'Ã©lÃ©ment Ã©tait prÃ©sent dans l'ensemble (au sens de la
     *         comparaison profonde) et qu'il a Ã©tÃ© retirÃ©, false sinon
     */
    public default boolean retrait(E element)
    {
        /*
         * TODO ComplÃ©ter ...
         */
        return false;
    }

    /**
     * Teste si l'ensemble est vide en utilisant l'itÃ©rateur ou bien le
     * {@link #cardinal}
     *
     * @return renvoie true si l'ensemble ne contient aucun Ã©lÃ©ment, false sinon
     * @see ensembles.Ensemble#estVide()
     * @note Attention, si l'on utilise cardinal dans estVide, il ne faut pas
     *       utiliser estVide dans cardinal et vice versa.
     */
    public default boolean estVide()
    {
        /*
         * TODO Remplacer par l'implÃ©mentation ...
         */
        return false;
    }

    /**
     * Test d'appartenance d'un Ã©lÃ©ment Ã  l'ensemble en utilisant l'itÃ©rateur
     * pour parcourir les Ã©lÃ©ments
     *
     * @param element l'Ã©lÃ©ment dont on doit tester l'appartenance
     * @return true si l'Ã©lÃ©ment est prÃ©sent dans l'ensemble (au sens de la
     *         comparaison profonde), false sinon
     */
    public default boolean contient(E element)
    {
        /*
         * TODO ComplÃ©ter ...
         */

        return false;
    }

    /**
     * Test si ensemble est un sous-ensemble de l'ensemble courant. C'est Ã  dire
     * si l'ensemble courant contient tous les Ã©lÃ©ments de l'ensemble passÃ© en
     * argument
     *
     * @note Si l'ensemble passÃ© en argument est null il ne sera pas considÃ©rÃ©
     *       comme contenu.
     * @param ensemble l'ensemble dont on veut tester s'il est un sous ensemble
     *          de l'ensemble courant
     * @return true si ensemble est un sous-ensemble de l'ensemble courant,
     *         false sinon. false si ensemble est null.
     */
    public default boolean contient(Ensemble<E> ensemble)
    {
```

```
91          /*
92           * TODO ComplÃ©ter ...
93           */
94
95          return false;
96      }
97
98      /**
99       * Efface tous les Ã©lements de l'ensemble en utilisant le remove de
100      * l'itÃ©rateur fournit par {@link #iterator()}
101      */
102     public default void efface()
103     {
104         /*
105          * TODO ComplÃ©ter ...
106          */
107     }
108
109     /**
110      * Taille de l'ensemble en utilisant l'itÃ©rateur
111      *
112      * @return le nombre d'Ã©lÃ©ments dans l'ensemble
113      * @see ensembles.Ensemble#cardinal() Attention : si l'on utilise estVide
114      *      dans cardinal, il ne faut pas utiliser cardinal dans estVide
115      * @note Cette mÃ©thode aura intÃ©rÃªt Ã  Ãªtre rÃ©implÃ©mentÃ©e dans les classes
116      *       filles qui utilisent des conteneurs pouvant donner leur taille
117      *       directement
118      */
119     public default int cardinal()
120     {
121         int count = 0;
122
123         /*
124          * TODO ComplÃ©ter ...
125          */
126
127         return count;
128     }
129
130     /**
131      * Union avec un autre ensemble : (this union ensemble).
132      *
133      * @param ensemble l'autre ensemble avec lequel on veut crÃ©er une union
134      * @return un nouvel ensemble contenant l'union de l'ensemble courant et de
135      *         l'ensemble passÃ© en argument
136      */
137     public abstract Ensemble<E> union(Ensemble<E> ensemble);
138
139     /**
140      * ImplÃ©mentation de classe de l'union de deux ensemble dans un autre
141      * ensemble
142      *
143      * @param ens1 le premier ensemble
144      * @param ens2 le second ensemble
145      * @param res l'ensemble contenant l'union de ens1 et ens2
146      */
147     public static <E> void union(Ensemble<E> ens1, Ensemble<E> ens2, Ensemble<E> res)
148     {
149         /*
150          * TODO ComplÃ©ter ...
151          */
152     }
153
154     /**
155      * Intersection avec un autre ensemble : (this inter ensemble).
156      *
157      * @param ensemble l'autre ensemble avec lequel on veut crÃ©er une
158      *            intersection
159      * @return un nouvel ensemble contenant l'intersection de l'ensemble courant
160      *         et de l'ensemble passÃ© en argument
161      */
162     public abstract Ensemble<E> intersection(Ensemble<E> ensemble);
163
164     /**
165      * ImplÃ©mentation de classe de l'intersection de deux ensemble dans un autre
166      * ensemble
167      *
168      * @param ens1 le premier ensemble
169      * @param ens2 le second ensemble
170      * @param res l'ensemble contenant l'intersection de ens1 et ens2
171      */
172     public static <E> void intersection(Ensemble<E> ens1, Ensemble<E> ens2, Ensemble<E> res)
173     {
174         /*
175          * TODO ComplÃ©ter ...
176          */
177     }
178
179     /**
180      * ComplÃ©ment avec un autre ensemble : (this - ensemble).
```



```
181      *
182      * @param ensemble l'autre ensemble avec lequel on veut crÃ©er le complÃ©ment
183      * @return un nouvel ensemble contenant uniquement les Ã©lÃ©ments prÃ©sents
184      *         dans l'ensemble courant mais PAS dans l'ensemble passÃ© en
185      *         argument
186      */
187     public abstract Ensemble<E> complement(Ensemble<E> ensemble);
188
189     /**
190      * ImplÃ©mentation de classe du complÃ©ment de deux ensembles dans un autre
191      * ensemble.
192      *
193      * @param ens1 le premier ensemble
194      * @param ens2 le second ensemble
195      * @param res l'ensemble contenant le complÃ©ment de ens1 - ens2
196      */
197     public static <E> void complement(Ensemble<E> ens1, Ensemble<E> ens2, Ensemble<E> res)
198     {
199         /*
200          * TODO ComplÃ©ter ...
201          */
202     }
203
204     /**
205      * DiffÃ©rence symmÃ©trique avec un autre ensemble : (this delta ensemble).
206      * L'ensemble correspondant Ã  la diffÃ©rence symÃ©trique contient les Ã©lÃ©ment
207      * qui sont soit dans l'ensemble courant, soit dans l'autre ensemble mais
208      * pas dans les deux ensembles = (this - ensemble) union (ensemble - this)
209      *
210      * @param ensemble l'autre ensemble avec lequel on veut crÃ©er une diffÃ©rence
211      *            symÃ©trique
212      * @return un nouvel ensemble contenant la diffÃ©rence symmÃ©trique de
213      *         l'ensemble courant et de l'ensemble passÃ© en argument
214      * @see ensembles.Ensemble#difference(ensembles.Ensemble)
215      */
216     public default Ensemble<E> difference(Ensemble<E> ensemble)
217     {
218         /*
219          * TODO Remplacer par l'implÃ©mentation en utilisant
220          *  - Soit (A - B) âM-^Hª (B - A)
221          *  - Soit (A âM-^Hª B) - (B âM-^H© A)
222          */
223         return null;
224     }
225
226     /**
227      * Type des Ã©lements de l'ensemble
228      *
229      * @return une instance de la classe Class reprÃ©sentant le type des Ã©lements
230      *         de l'ensemble si celui ci n'est pas vide, ou bien null si
231      *         l'ensemble est vide.
232      * @note cette mÃ©thode sera utile dans l'implÃ©mentation de la mÃ©thode
233      *       {@link #equals(Object)} pour dÃ©terminer si deux ensembles ont le
234      *       mÃªme type d'Ã©lÃ©ments
235      * @see ensembles.Ensemble#typeElements()
236      */
237     @SuppressWarnings("unchecked")
238     public default Class<E> typeElements()
239     {
240         Iterator<E> it = iterator();
241         if (it ≠ null)
242         {
243             if (it.hasNext())
244             {
245                 return (Class<E>) it.next().getClass();
246             }
247         }
248
249         return null;
250     }
251
252     // ----------------------------------------------------------------------
253     // MÃ©thodes Ã  implÃ©menter dÃ©finies dans la classe Object
254     // ----------------------------------------------------------------------
255
256     /**
257      * Test d'Ã©galitÃ© entre deux ensembles
258      *
259      * @param o l'objet Ã  comparer
260      * @return true si l'objet Ã  comparer est un ensemble et qu'il contient les
261      *         mÃªmes Ã©lÃ©ments (pas forcÃ©ment dans le mÃªme ordre). Si les deux
262      *         ensembles sont vides on considÃ¨re qu'ils seront Ã©gaux quel que
263      *         soit leur type de contenu (dans la mesure oÃ¹ l'on ne peut pas le
264      *         dÃ©terminer avec {@link ensembles.Ensemble#typeElements()}
265      * @note une interface ne peut pas implÃ©menter par dÃ©faut des mÃ©thodes
266      *       surchargÃ©es de la classe object (celles ci dÃ©pendant de l'Ã©tat
267      *       interne des objets, ce qui n'est pas le cas d'une interface)
268      */
269     @Override
270     public abstract boolean equals(Object o);
```

```
271
272     /**
273      * Hashcode d'un ensemble. Le HashCode d'un ensemble doit Ãªtre calculÃ© comme
274      * Ã©tant la somme des hascodes de ses Ã©lÃ©ments afin de ne pas tenir compte
275      * de l'ordre des Ã©lÃ©ments dans la collection sous-jacente.
276      *
277      * @return le hashage d'un ensemble
278      * @note une interface ne peut pas implÃ©menter par dÃ©faut des mÃ©thodes
279      *       surchargÃ©es de la classe object (celles ci dÃ©pendant de l'Ã©tat
280      *       interne des objets, ce qui n'est pas le cas d'une interface)
281      */
282     @Override
283     public abstract int hashCode();
284
285     /**
286      * Affichage des Ã©lement de l'ensemble sous la forme : par exemple pour un
287      * ensemble de 3 elts : "[elt1, elt2, elt3]" oÃ¹ eltn reprÃ©sente le toString
288      * du niÃ¨me elt.
289      *
290      * @return une chaine de caractÃ¨re reprÃ©sentant les Ã©lÃ©ments de l'ensemble
291      *         sÃ©parÃ© par des virgules et encadrÃ© par des crochets
292      * @note une interface ne peut pas implÃ©menter par dÃ©faut des mÃ©thodes
293      *       surchargÃ©es de la classe object (celles ci dÃ©pendant de l'Ã©tat
294      *       interne des objets, ce qui n'est pas le cas d'une interface)
295      */
296     @Override
297     public abstract String toString();
298
299     // ------------------------------------------------------------------------
300     // MÃ©thodes Ã  implÃ©menter dÃ©finies dans l'interface Iterable<E>
301     // ------------------------------------------------------------------------
302     /**
303      * Factory method fournissant un itÃ©rateur sur l'ensemble
304      *
305      * @return un nouvel itÃ©rateur sur cet ensemble
306      */
307     @Override
308     public abstract Iterator<E> iterator();
309 }
```



```
1   package ensembles;
2
3   import java.util.Iterator;
4
5   /**
6    * Ensemble GÃ©nÃ©rique implÃ©mentant partiellement les opÃ©rations communes Ã  tous
7    * les ensembles quels que soit les conteneurs sous-jacents utilisÃ©s pour
8    * stocker les Ã©lÃ©ments de l'ensemble. L'ensemble gÃ©nÃ©rique est implÃ©mentÃ© en
9    * majeure partie grÃ¢ce Ã  l'itÃ©rateur fourni par la mÃ©thode {@link #iterator()}
10   *
11   * @author davidroussel
12   */
13  public abstract class EnsembleGenerique<E> implements Ensemble<E>
14  {
15      /*
16       * (non-Javadoc)
17       * @see ensembles.Ensemble#ajout(java.lang.Object)
18       */
19      @Override
20      public abstract boolean ajout(E element);
21
22      /*
23       * (non-Javadoc)
24       * @see ensembles.Ensemble#union(ensembles.Ensemble)
25       */
26      @Override
27      public abstract Ensemble<E> union(Ensemble<E> ensemble);
28
29      /*
30       * (non-Javadoc)
31       * @see ensembles.Ensemble#intersection(ensembles.Ensemble)
32       */
33      @Override
34      public abstract Ensemble<E> intersection(Ensemble<E> ensemble);
35
36      /*
37       * (non-Javadoc)
38       * @see ensembles.Ensemble#complement(ensembles.Ensemble)
39       */
40      @Override
41      public abstract Ensemble<E> complement(Ensemble<E> ensemble);
42
43      /*
44       * (non-Javadoc)
45       * @see ensembles.Ensemble#iterator()
46       */
47      @Override
48      public abstract Iterator<E> iterator();
49
50      /**
51       * Test d'Ã©galitÃ© entre deux ensembles
52       *
53       * @param o l'objet Ã  comparer
54       * @return true si l'objet Ã  comparer est un ensemble et qu'il contient les
55       *         mÃªmes Ã©lÃ©ments (pas forcÃ©ment dans le mÃªme ordre). Si les deux
56       *         ensembles sont vides on considÃ¨re qu'ils seront Ã©gaux quel que
57       *         soit leur type de contenu (dans la mesure oÃ¹ l'on ne peut pas le
58       *         dÃ©terminer avec {@link ensembles.Ensemble#typeElements()}
59       * @see java.lang.Object#equals(java.lang.Object)
60       */
61      @Override
62      public boolean equals(Object obj)
63      {
64          /*
65           * TODO Remplacer par :
66           *  1 - obj == null ? ==> false
67           *  2 - obj == this ? ==> true
68           *  3 - obj est une instance de Ensemble<?> ?
69           *      - caster obj en Ensemble<?>
70           *          - les typeElements() sont identiques ?
71           *              - si typeElements des 2 est null :
72           *                  ensembles vides ==> true
73           *              - sinon - caster obj en (Ensemble<E>)
74           *                  - si tous les elts de l'un sont contenus dans l'autre ==> true
75           *                  - sinon ==> false
76           *          - sinon (types Ã©lÃ©ments diffÃ©rents) ==> false
77           *  - sinon obj n'est pas une instance de Ensemble<?> ==> false
78           */
79          return false;
80      }
81
82      /**
83       * Hashcode d'un ensemble en utilisant l'itÃ©rateur pour parcourir les
84       * Ã©lÃ©ments. Le HashCode d'un ensemble doit Ãªtre calculÃ© comme Ã©tant la
85       * somme des hashcodes de ses Ã©lÃ©ments afin de ne pas tenir compte de
86       * l'ordre des Ã©lÃ©ments dans la collection sous-jacente.
87       *
88       * @return le hashage d'un ensemble
89       * @see java.lang.Object#hashCode()
90       */
```

```
91      @Override
92      public int hashCode()
93      {
94          int result = 0;
95          /*
96           * TODO ComplÃ©ter ...
97           */
98          return result;
99      }
100
101     /**
102      * Affichage des Ã©lement de l'ensemble sous la forme : par exemple pour un
103      * ensemble de 3 elts : "[elt1, elt2, elt3]" oÃ¹ eltn reprÃ©sente le toString
104      * du niÃ¨me elt.
105      *
106      * @return une chaine de caractÃ¨re reprÃ©sentant les Ã©lÃ©ments de l'ensemble
107      *         sÃ©parÃ© par des virgules et encadrÃ© par des crochets
108      * @see java.lang.Object#toString()
109      */
110     @Override
111     public String toString()
112     {
113         StringBuilder sb = new StringBuilder();
114         sb.append("[");
115         /*
116          * TODO ComplÃ©ter ...
117          */
118         sb.append("]");
119
120         return new String(sb);
121     }
122 }
```



```
1   package ensembles;
2
3   import java.util.Iterator;
4
5   import tableaux.Tableau;
6
7   /**
8    * Ensemble Ã  base de tableaux
9    *
10   * @author davidroussel
11   */
12  public class EnsembleTableau<E> extends EnsembleGenerique<E>
13  {
14      /**
15       * Conteneur sous-jacent : un Tableau<E>
16       */
17      protected Tableau<E> tableau;
18
19      /**
20       * Constructeur par dÃ©faut d'un ensemble Ã  base de {@link tableaux.Tableau}
21       */
22      public EnsembleTableau()
23      {
24          /*
25           * TODO Remplacer par l'initialisation du tableau
26           */
27          tableau = null;
28      }
29
30      /**
31       * Constructeur de copie Ã  partir d'un {@link Iterable}
32       *
33       * @param elements l'itÃ©rable dont on doit copier les Ã©lÃ©ments
34       */
35      public EnsembleTableau(Iterable<E> elements)
36      {
37          /*
38           * TODO Remplacer par l'initialisation du tableau, puis l'ajout (au
39           * sens des ensembles) des Ã©lÃ©ments de "elements"
40           */
41          tableau = null;
42      }
43
44      /**
45       * Ajout d'un Ã©lÃ©ment Ã  un ensemble ssi celui ci n'est pas null et qu'il
46       * n'est pas dÃ©jÃ  prÃ©sent
47       * Ce qui revient dans le cas prÃ©sent Ã  ajouter un Ã©lÃ©ment au tableau si
48       * celui ci n'y est pas dÃ©jÃ  prÃ©sent
49       *
50       * @param element l'Ã©lÃ©ment Ã  ajouter Ã  l'ensemble (on considÃ¨rera que l'on
51       *                ne peut pas ajouter d'Ã©lÃ©ment null)
52       * @return true si l'Ã©lÃ©ment a pu Ãªtre ajoutÃ© Ã  l'ensemble, false sinon ou
53       *         si l'on a tentÃ© d'insÃ©rer un Ã©lÃ©ment null (auquel cas il n'est
54       *         pas insÃ©rÃ©)
55       * @see ensembles.EnsembleGenerique#ajout(java.lang.Object)
56       */
57      @Override
58      public boolean ajout(E element)
59      {
60          /*
61           * TODO ComplÃ©ter ...
62           */
63          return false;
64      }
65
66      /**
67       * Taille de l'ensemble : rÃ©implÃ©mentation en utilisant les propriÃ©tÃ©s du
68       * tableau sous-jacent plutÃ´t que l'itÃ©rateur (amÃ©lioration de performances)
69       *
70       * @return le nombre d'Ã©lÃ©ments dans l'ensemble
71       * @see ensembles.EnsembleGenerique#cardinal()
72       */
73      @Override
74      public int cardinal()
75      {
76          /*
77           * TODO Remplacer par une implÃ©mentation plus performante que celle
78           * fournie par dÃ©faut par l'interface Ensemble<E>
79           */
80          return 0;
81      }
82
83      /**
84       * Union avec un autre ensemble en rÃ©utilisant la mÃ©thode de classe union
85       * Ã©crite dans l'ensemble GÃ©nÃ©rique (
86       * {@link ensembles.EnsembleGenerique#union(ensembles.Ensemble, ensembles.Ensemble, ensembles.En
    semble)}
87       * ) et un nouvel {@link ensemble.EnsembleTableau} pour stocker le rÃ©sultat.
88       *
89       * @param ensemble l'autre ensemble avec lequel on veut crÃ©er une union
```

```
90      * @return un nouvel ensemble contenant l'union de l'ensemble courant et de
91      *         l'ensemble passÃ© en argument
92      * @see ensembles.EnsembleGenerique#union(ensembles.Ensemble,
93      *      ensembles.Ensemble, ensembles.Ensemble)
94      */
95     @Override
96     public Ensemble<E> union(Ensemble<E> ensemble)
97     {
98         /*
99          * TODO Remplacer par :
100         *  - la crÃ©ation d'un nouvel ensemble rÃ©sultat
101         *  - l'union de this et ensemble dans rÃ©sultat en rÃ©utilisant ce que
102         *    l'on a dÃ©jÃ  Ã©crit
103         *  - le renvoi de rÃ©sultat
104         */
105        return null;
106    }
107
108    /**
109     * Intersection avec un autre ensemble en rÃ©utilisant la mÃ©thode de classe
110     * intersection Ã©crite dans l'ensemble GÃ©nÃ©rique (
111     * {@link ensembles.EnsembleGenerique#intersection(ensembles.Ensemble, ensembles.Ensemble, ensem
    bles.Ensemble)}
112     * ) et un nouvel {@link ensemble.EnsembleTableau} pour stocker le rÃ©sultat.
113     *
114     * @param ensemble l'autre ensemble avec lequel on veut crÃ©er une
115     *             intersection
116     * @return un nouvel ensemble contenant l'intersection de l'ensemble courant
117     *         et de l'ensemble passÃ© en argument
118     * @see ensembles.EnsembleGenerique#intersection(ensembles.Ensemble,
119     *      ensembles.Ensemble, ensembles.Ensemble)
120     */
121    @Override
122    public Ensemble<E> intersection(Ensemble<E> ensemble)
123    {
124        /*
125         * TODO Remplacer par :
126         *  - la crÃ©ation d'un nouvel ensemble rÃ©sultat
127         *  - l'intersection de this et ensemble dans rÃ©sultat en rÃ©utilisant
128         *    ce que l'on a dÃ©jÃ  Ã©crit
129         *  - le renvoi de rÃ©sultat
130         */
131        return null;
132    }
133
134    /**
135     * ComplÃ©ment avec un autre ensemble en rÃ©utilisant la mÃ©thode de classe
136     * complement Ã©crite dans l'ensemble GÃ©nÃ©rique (
137     * {@link ensembles.EnsembleGenerique#complement(ensembles.Ensemble, ensembles.Ensemble, ensembl
    es.Ensemble)}
138     * ) et un nouvel {@link ensemble.EnsembleTableau} pour stocker le rÃ©sultat.
139     *
140     * @param ensemble l'autre ensemble avec lequel on veut crÃ©er le complÃ©ment
141     * @return un nouvel ensemble contenant uniquement les Ã©lÃ©ments prÃ©sents
142     *         dans l'ensemble courant mais PAS dans l'ensemble passÃ© en
143     *         argument
144     * @see ensembles.EnsembleGenerique#complement(ensembles.Ensemble,
145     *      ensembles.Ensemble, ensembles.Ensemble)
146     */
147    @Override
148    public Ensemble<E> complement(Ensemble<E> ensemble)
149    {
150        /*
151         * TODO Remplacer par :
152         *  - la crÃ©ation d'un nouvel ensemble rÃ©sultat
153         *  - le complÃ©ment de this et ensemble dans rÃ©sultat en rÃ©utilisant
154         *    ce que l'on a dÃ©jÃ  Ã©crit
155         *  - le renvoi de rÃ©sultat
156         */
157        return null;
158    }
159
160    /**
161     * Factory method fournissant un itÃ©rateur sur l'ensemble en rÃ©utilisant
162     * l'itÃ©rateur du tableau sous-jacent
163     *
164     * @return un nouvel itÃ©rateur sur cet ensemble
165     * @see ensembles.EnsembleGenerique#iterator()
166     */
167    @Override
168    public Iterator<E> iterator()
169    {
170        /*
171         * TODO Remplacer par la crÃ©ation d'un iterateur du tableau
172         */
173        return null;
174    }
175 }
```



```
1  package ensembles;
2
3  import java.lang.reflect.Constructor;
4  import java.lang.reflect.InvocationTargetException;
5
6  /**
7   * Factory permettant de crÃ©er diffÃ©rents types d'ensembles utilisÃ©s dans les
8   * tests
9   *
10  * @author davidroussel
11  */
12 public class EnsembleFactory<E>
13 {
14     /**
15      * Obtention d'un nouvel ensemble d'aprÃ¨s le type d'ensemble souhaitÃ© et un
16      * contenu (Ã©ventuel) Ã  copier dans le nouvel ensemble
17      *
18      * @param typeEnsemble le type d'ensemble demandÃ©: soit
19      *             {@link ensembles.EnsembleVector}, soit
20      *             {@link ensembles.EnsembleListe}, soit
21      *             {@link ensembles.EnsembleTableau}
22      * @param content le contenu Ã©ventuel Ã  copier dans le nouvel ensemble ( si
23      *             celui ci est nul le constructeur par dÃ©faut sera appelÃ©, s'il
24      *             est non null, le constructeur de copie sera appelÃ©
25      * @return une nouvelle instance de l'ensemble correspondant au type demandÃ©
26      * @throws SecurityException Si le SecurityManager ne permet pas l'accÃ¨s au
27      *             constructeur demandÃ©
28      * @throws NoSuchMethodException Si le constructeur demandÃ© n'existe pas
29      * @throws IllegalArgumentException Si le nombre d'arguments fournis au
30      *             constructeur n'est pas le bon
31      * @throws InstantiationException si la classe demandÃ©e est abstraite
32      * @throws IllegalAccessException Si le constructeur demandÃ© est
33      *             inaccessible
34      * @throws InvocationTargetException si le constructeur invoquÃ© dÃ©clenche
35      *             une exception
36      */
37     @SuppressWarnings("unchecked")
38     public static <E> Ensemble<E> getEnsemble(Class<? extends Ensemble<E>> typeEnsemble, Iterable<E>
    content)
39             throws SecurityException, NoSuchMethodException, IllegalArgumentException, Instantiation
    Exception,
40             IllegalAccessException, InvocationTargetException
41     {
42         Constructor<? extends Ensemble<E>> constructor = null;
43         Class<?>[] argumentsTypes = null;
44         Object[] arguments = null;
45         Object instance = null;
46
47         if (content ≡ null)
48         {
49             argumentsTypes = new Class<?>[0];
50             arguments = new Object[0];
51         }
52         else
53         {
54             argumentsTypes = new Class<?>[1];
55             argumentsTypes[0] = Iterable.class;
56             arguments = new Object[1];
57             arguments[0] = content;
58         }
59
60         constructor = typeEnsemble.getConstructor(argumentsTypes);
61
62         if (constructor ≠ null)
63         {
64             instance = constructor.newInstance(arguments);
65         }
66
67         return (Ensemble<E>) instance;
68     }
69 }
```

```java
package ensembles;

import java.util.Collection;

/**
 * Ensemble d'Ã©lements triÃ©s. Les Ã©lements doivent donc Ãªtre des
 * {@link Comparable} afin de pouvoir rÃ©aliser l'insertion triÃ©e de nouveaux
 * Ã©lÃ©ments dans {@link #ajout(Comparable)}. A titre d'information les
 * {@link Integer} et les {@link String} sont des {@link Comparable}.
 *
 * @author davidroussel
 */
public interface EnsembleTri<E extends Comparable<E>> extends Ensemble<E>
{
    /*
     * Note : les redÃ©finitions ci-dessous ne sont pas techniquement nÃ©cessaires
     * (sauf rang()) mais permettent de documenter les changements nÃ©cessaires
     * dans la rÃ©implÃ©mentation de ces mÃ©thodes spÃ©cifiquement pour les
     * ensembles triÃ©s.
     */

    /**
     * Ajout d'un nouvel Ã©lÃ©ment de maniÃ¨re Ã  maintenir l'ensemble triÃ©
     *
     * @param element l'Ã©lÃ©ment Ã  ajouter de maniÃ¨re triÃ©e
     * @return true si l'Ã©lÃ©ment n'Ã©tait pas dÃ©jÃ  prÃ©sent dans l'ensemble, false
     *         sinon.
     */
    @Override
    public abstract boolean ajout(E element);

    /**
     * Code de hachage d'un ensemble triÃ©. Il est nÃ©cessaire de rÃ©implÃ©menter le
     * code de hachage pour les ensembles triÃ©s car on considÃ¨rera que deux
     * ensembles contenant les mÃªme Ã©lÃ©ments mais dans des ordres diffÃ©rents
     * seront eux-mÃªme diffÃ©rents. Il faut donc que la mÃ©thode hashCode prenne
     * en compte l'ordre des Ã©lÃ©ments (Comme dans les autres {@link Collection}
     * d'ailleurs).
     *
     * @return le code de hachage de cet ensemble triÃ©.
     * @see listes.Liste#hashCode() tableaux.Tableau#hashCode() pour un exemple
     *      de hashage utilisant l'ordre des Ã©lÃ©ments
     */
    @Override
    public abstract int hashCode();

    /**
     * Test d'Ã©galitÃ© d'un ensemble triÃ©. Il est nÃ©cessaire de rÃ©implÃ©menter la
     * comparaison avec un autre ensemble car l'ordre des Ã©lÃ©ment aura son
     * importance dans la comparaison ce qui n'Ã©tait pas le cas avec les
     * ensembles non triÃ©s.
     *
     * @return true si l'objet obj est aussi un ensemble (pas forcÃ©ment triÃ©) et
     *         qu'il contient exactement les mÃªmes Ã©lÃ©ments dans le mÃªme ordre.
     */
    @Override
    public abstract boolean equals(Object obj);

    /**
     * Calcule le rang oÃ¹ doit Ãªtre insÃ©rÃ© un Ã©lÃ©ment de maniÃ¨re triÃ©e dans
     * l'ensemble triÃ©
     *
     * @param Element l'Ã©lÃ©ment dont on veut calculer le rang dans l'ensemble
     *            triÃ©
     * @return le rang d'insertion de l'Ã©lÃ©ment dans l'ensemble triÃ©
     */
    public default int rang(E element)
    {
        /*
         * calcul du rang d'un nouvel Ã©lÃ©ment : On parcours les Ã©lÃ©ments de this
         * et si un elt de this est plus grand que l'element Ã  inserer ((elt de
         * this).compareTo(element) >= 0) on a trouvÃ© le rang oÃ¹ insÃ©rer, on
         * quitte alors la boucle sans passer au suivant et on renvoie le nombre
         * d'itÃ©rations effectuÃ©es. Cas limites : - element < 1er elt de this on
         * quitte la boucle immÃ©diatement - element > dernier elt de this la
         * boucle va jusqu'au bout
         */
        int res = 0;
        /*
         * TODO ComplÃ©ter ...
         */
        return res;
    }
}
```



```java
package ensembles;

import java.util.Collection;

import tableaux.Tableau;

/**
 * Ensemble triÃ© utilisant un {@link Tableau}
 *
 * @author davidroussel
 */
public class EnsembleTriTableau<E extends Comparable<E>> extends
        EnsembleTableau<E> implements EnsembleTri<E>
{

    /**
     * Constructeur par dÃ©faut d'un ensemble triÃ© utilisant un {@link Tableau}
     */
    public EnsembleTriTableau()
    {
        /*
         * TODO ComplÃ©ter si besoin ...
         */
    }

    /**
     * Constructeur de copie Ã  partir d'un autre iterable
     *
     * @param elements l'itÃ©rable dont on veut copier les Ã©lÃ©ments
     */
    public EnsembleTriTableau(Iterable<E> elements)
    {
        /*
         * TODO ComplÃ©ter ...
         */
    }

    /**
     * Ajout d'un Ã©lÃ©ment de maniÃ¨re triÃ©e dans l'ensemble utilisant un
     * {@link Tableau}
     * @param element l'Ã©lÃ©ment Ã  ajouter de maniÃ¨re triÃ©e (on considÃ¨rera que
     *            l'on ne peut pas ajouter d'Ã©lÃ©ment null)
     * @return true si l'Ã©lÃ©ment n'Ã©tait pas dÃ©jÃ  prÃ©sent dans l'ensemble, false
     *         sinon ou si l'on a tentÃ© d'insÃ©rer un Ã©lÃ©ment null (auquel cas il
     *         n'est pas insÃ©rÃ©).
     * @see ensembles.EnsembleTableau#ajout(java.lang.Object)
     * @see tableaux.Tableau#insertElement(E, int)
     */
    @Override
    public boolean ajout(E element)
    {
        /*
         * TODO ComplÃ©ter ...
         */
        return false;
    }

    /**
     * Test d'Ã©galitÃ© d'un ensemble triÃ©. Il est nÃ©cessaire de rÃ©implÃ©menter la
     * comparaison avec un autre ensemble car l'ordre des Ã©lÃ©ment aura son
     * importance dans la comparaison ce qui n'Ã©tait pas le cas avec les
     * ensembles non triÃ©s.
     *
     * @return true si l'objet obj est aussi un ensemble (pas forcÃ©ment triÃ©) et
     *         qu'il contient exactement les mÃªmes Ã©lÃ©ments dans le mÃªme ordre.
     * @see ensembles.EnsembleGenerique#equals(java.lang.Object)
     */
    @Override
    public boolean equals(Object obj)
    {
        /*
         * TODO Remplacer par ...
         *  1 - obj == null ? ==> false
         *  2 - obj == this ? ==> true
         *  3 - obj est une instance de Ensemble<?>
         *      - caster obj en Ensemble<?>
         *      - si obj et this ont exactement les mÃªmes Ã©lÃ©ments dans le
         *        mÃªme ordre ==> true
         *      - sinon ==> false;
         *    - sinon (obj n'est pas un Ensembble<?>) ==> false
         */
        return false;
    }

    /**
     * Code de hachage d'un ensemble triÃ©. Il est nÃ©cessaire de rÃ©implÃ©menter le
     * code de hachage pour les ensembles triÃ©s car on considÃ¨rera que deux
     * ensembles contenant les mÃªme Ã©lÃ©ments mais dans des ordres diffÃ©rents
     * seront eux-mÃªme diffÃ©rents. Il faut donc que la mÃ©thode hashCode prenne
     * en compte l'ordre des Ã©lÃ©ments (Comme dans les autres {@link Collection}
```

```
91       * d'ailleurs).
92       *
93       * @return le code de hachage de cet ensemble triÃ©.
94       * @see tableaux.Tableau#hashCode() pour un exemple de hashage utilisant
95       *      l'ordre des Ã©lÃ©ments
96       * @see ensembles.EnsembleGenerique#hashCode()
97       */
98      @Override
99      public int hashCode()
100     {
101         final int prime = 31;
102         int result = 1;
103         /*
104          * TODO ComplÃ©ter ...
105          */
106         return result;
107     }
108 }
```



```
1  package ensembles;
2
3  import java.util.Collection;
4  import java.util.Iterator;
5
6  /**
7   * ImplÃ©mentation gÃ©nÃ©rique partielle d'un ensemble triÃ© sous forme de
8   * dÃ©corateur d'un ensemble ordinaire.
9   *
10  * @author davidroussel
11  */
12 public abstract class EnsembleTriGenerique<E extends Comparable<E>>
13 extends EnsembleGenerique<E> implements EnsembleTri<E>
14 {
15     /**
16      * Ensemble de base sous-jacent dÃ©corÃ© par les ensembles triÃ©s.
17      */
18     protected Ensemble<E> ensemble;
19
20     /**
21      * Ajout d'un nouvel Ã©lÃ©ment de maniÃ¨re Ã  maintenir l'ensemble triÃ© en
22      * utilisant la mÃ©thode {@link #insererAuRang(E element, int rang)} ssi
23      * l'Ã©lÃ©ment peut Ãªtre insÃ©rÃ© dans cet ensemble triÃ©
24      *
25      * @param element l'Ã©lÃ©ment Ã  ajouter de maniÃ¨re triÃ©e (on considÃ¨rera que
26      *            l'on ne peut pas ajouter d'Ã©lÃ©ment null)
27      * @return true si l'Ã©lÃ©ment n'Ã©tait pas dÃ©jÃ  prÃ©sent dans l'ensemble, false
28      *         sinon ou si l'on a tentÃ© d'insÃ©rer un Ã©lÃ©ment null (auquel cas il
29      *         n'est pas insÃ©rÃ©).
30      * @see ensembles.EnsembleListe#ajout(java.lang.Object)
31      */
32     @Override
33     public boolean ajout(E element)
34     {
35         /*
36          * TODO ComplÃ©ter ...
37          */
38         return false;
39     }
40
41     /**
42      * Insertion d'un nouvel Ã©lÃ©ment au rang choisi en utilisant
43      * {@link #rang(E element)} pour calculer le rang d'insertion de l'Ã©lÃ©ment
44      *
45      * @param element l'Ã©lement Ã  insÃ©rer
46      * @param rang le rang oÃ¹ insÃ©rer cet Ã©lÃ©ment
47      * @return true si l'Ã©lÃ©ment a Ã©tÃ© insÃ©rÃ© au rang choisi, false si l'Ã©lement
48      *         n'a pas pu Ãªtre insÃ©rÃ© Ã  cause d'un rang invalide
49      * @note On remarquera que la mÃ©thode ne teste pas au prÃ©alable l'existence
50      *       de l'Ã©lÃ©ment Ã  insÃ©rer dans l'ensemble car c'est la mÃ©thode
51      *       {@link #ajout(E)} qui s'en chargera
52      */
53     protected abstract boolean insererAuRang(E element, int rang);
54
55     /**
56      * Test d'Ã©galitÃ© d'un ensemble triÃ©. Il est nÃ©cessaire de rÃ©implÃ©menter la
57      * comparaison avec un autre ensemble car l'ordre des Ã©lÃ©ment aura son
58      * importance dans la comparaison ce qui n'Ã©tait pas le cas avec les
59      * ensembles non triÃ©s.
60      *
61      * @return true si l'objet obj est aussi un ensemble (pas forcÃ©ment triÃ©) et
62      *         qu'il contient exactement les mÃªmes Ã©lÃ©ments dans le mÃªme ordre.
63      * @see ensembles.EnsembleGenerique#equals(java.lang.Object)
64      */
65     @Override
66     public boolean equals(Object obj)
67     {
68         /*
69          * TODO Remplacer par ...
70          *  1 - obj == null ? ==> false
71          *  2 - obj == this ? ==> true
72          *  3 - obj est une instance de Ensemble<?>
73          *      - caster obj en Ensemble<?>
74          *          - si obj et this ont exactement les mÃªmes Ã©lÃ©ments dans le mÃªme ordre ==> tr
    ue
75          *          - sinon ==> false;
76          *  - sinon (obj n'est pas un Ensemble<?>) ==> false
77          */
78         return false;
79     }
80
81     /**
82      * Code de hachage d'un ensemble triÃ©. Il est nÃ©cessaire de rÃ©implÃ©menter le
83      * code de hachage pour les ensembles triÃ©s car on considÃ¨rera que deux
84      * ensembles contenant les mÃªme Ã©lÃ©ments mais dans des ordres diffÃ©rents
85      * seront eux-mÃªme diffÃ©rents. Il faut donc que la mÃ©thode hashCode prenne
86      * en compte l'ordre des Ã©lÃ©ments (Comme dans les autres {@link Collection}
87      * d'ailleurs).
88      *
89      * @return le code de hachage de cet ensemble triÃ©.
```

```
90       * @see listes.Liste#hashCode() ou tableaux.Tableau#hashCode() pour un
91       *      exemple de hashage utilisant l'ordre des Ã©lÃ©ments
92       * @see ensembles.EnsembleGenerique#hashCode()
93       */
94      @Override
95      public int hashCode()
96      {
97          final int prime = 31;
98          int result = 1;
99          /*
100          * TODO ComplÃ©ter ...
101          */
102         return result;
103     }
104
105     /**
106      * Union avec un autre ensemble : reste semblable Ã  l'union avec avec un
107      * ensemble non triÃ© mais s'applique sur l'ensemble dÃ©corÃ© {@link #ensemble}
108      *
109      * @param ensemble l'autre ensemble avec lequel on veut crÃ©er une union
110      * @return un nouvel ensemble contenant l'union de l'ensemble courant et de
111      *         l'ensemble passÃ© en argument
112      * @see ensembles.EnsembleGenerique#union(ensembles.Ensemble)
113      */
114     @Override
115     public Ensemble<E> union(Ensemble<E> autre)
116     {
117         /*
118          * TODO Remplacer par l'implÃ©mentation ...
119          */
120         return null;
121     }
122
123     /**
124      * Intersection avec un autre ensemble : reste semblable Ã  l'intersection
125      * avec avec un ensemble non triÃ© mais s'applique sur l'ensemble dÃ©corÃ©
126      * {@link #ensemble}
127      *
128      * @param ensemble l'autre ensemble avec lequel on veut crÃ©er une
129      *                 intersection
130      * @return un nouvel ensemble contenant l'intersection de l'ensemble courant
131      *         et de l'ensemble passÃ© en argument
132      * @see ensembles.EnsembleGenerique#intersection(ensembles.Ensemble)
133      */
134     @Override
135     public Ensemble<E> intersection(Ensemble<E> autre)
136     {
137         /*
138          * TODO Remplacer par l'implÃ©mentation ...
139          */
140         return null;
141     }
142
143     /**
144      * Complement avec un autre ensemble : reste semblable au complement avec
145      * avec un ensemble non triÃ© mais s'applique sur l'ensemble dÃ©corÃ©
146      * {@link #ensemble}
147      *
148      * @param ensemble l'autre ensemble avec lequel on veut crÃ©er un complement
149      * @return un nouvel ensemble contenant le complement de l'ensemble courant
150      *         et de l'ensemble passÃ© en argument
151      * @see ensembles.EnsembleGenerique#complement(ensembles.Ensemble)
152      */
153     @Override
154     public Ensemble<E> complement(Ensemble<E> autre)
155     {
156         /*
157          * TODO Remplacer par l'implÃ©mentation ...
158          */
159         return null;
160     }
161
162     /**
163      * Factory method fournissant un itÃ©rateur sur l'ensemble en rÃ©utilisant
164      * l'itÃ©rateur de l'ensemble ordinaire sous-jacent.
165      *
166      * @return un nouvel itÃ©rateur sur cet ensemble
167      * @see ensembles.EnsembleGenerique#iterator()
168      */
169     @Override
170     public Iterator<E> iterator()
171     {
172         /*
173          * TODO Remplacer par l'implÃ©mentation ...
174          */
175         return null;
176     }
177 }
```



```
1   package ensembles;
2
3   import java.lang.reflect.InvocationTargetException;
4
5   /**
6    * Factory permettant de crÃ©er diffÃ©rents types d'ensembles triÃ©s utilisÃ©s dans
7    * les tests
8    *
9    * @author davidroussel
10   */
11  public class EnsembleTriFactory<E extends Comparable<E>>
12  {
13      /**
14       * Obtention d'un nouvel ensemble triÃ© d'aprÃ¨s le type d'ensemble souhaitÃ©
15       * et un contenu (Ã©ventuel) Ã  copier dans le nouvel ensemble
16       *
17       * @param typeEnsemble le type d'ensemble demandÃ©: soit
18       *            {@link ensembles.EnsembleTriVector}, soit
19       *            {@link ensembles.EnsembleTriVector2}, soit
20       *            {@link ensembles.EnsembleTriListe}, soit
21       *            {@link ensembles.EnsembleTriListe2}, soit
22       *            {@link ensembles.EnsembleTriTableau}, soit
23       *            {@link ensembles.EnsembleTriTableau2}
24       * @param content le contenu Ã©ventuel Ã  copier dans le nouvel ensemble ( si
25       *            celui ci est nul le constructeur par dÃ©faut sera appelÃ©, s'il
26       *            est non null, le constructeur de copie sera appelÃ©
27       * @return une nouvelle instance de l'ensemble correspondant au type demandÃ©
28       * @throws SecurityException Si le SecurityManager ne permet pas l'accÃ¨s au
29       *             constructeur demandÃ©
30       * @throws NoSuchMethodException Si le constructeur demandÃ© n'existe pas
31       * @throws IllegalArgumentException Si le nombre d'arguments fournis au
32       *             constructeur n'est pas le bon
33       * @throws InstantiationException si la classe demandÃ©e est abstraite
34       * @throws IllegalAccessException Si le constructeur demandÃ© est
35       *             inaccessible
36       * @throws InvocationTargetException si le constructeur invoquÃ© dÃ©clenche
37       *             une exception
38       */
39      public static <E extends Comparable<E>> EnsembleTri<E> getEnsemble(Class<? extends EnsembleTri<E
    >> typeEnsemble,
40              Iterable<E> content) throws SecurityException, NoSuchMethodException, IllegalArgumentExc
    eption,
41                      InstantiationException, IllegalAccessException, InvocationTargetException
42      {
43          return (EnsembleTri<E>) EnsembleFactory.<E> getEnsemble(typeEnsemble, content);
44      }
45  }
```

03 nov 13 19:24 **package−info.java** Page 1/1

```java
/**
 * Package contenant les classes de test
 */
package tests;
```

01 oct 14 17:15 **AllTests.java** Page 1/1

```java
package tests;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

/**
 * Suite de tests
 * @author davidroussel
 */
@RunWith(Suite.class)
@SuiteClasses(
{
    AllEnsembleTest.class,
    EnsembleTriTest.class
}
)
public class AllTests
{
    // Nothing
}
```

```java
package tests;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;

import java.lang.reflect.InvocationTargetException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

import ensembles.Ensemble;
import ensembles.EnsembleFactory;
import ensembles.EnsembleTableau;
import ensembles.EnsembleTri;

/**
 * Classe de test pour tous les types d'ensembles :
 * {@link ensembles.EnsembleVector}, {@link ensembles.EnsembleListe},
 * {@link ensembles.EnsembleTableau}.
 * Mais aussi pour les mÃ©thodes communes avec les ensemble triÃ©s tels que
 * {@link ensembles.EnsembleTriVector}, {@link ensembles.EnsembleTriVector2},
 * {@link ensembles.EnsembleTriListe}, {@link ensembles.EnsembleTriListe2},
 * {@link ensembles.EnsembleTriTableau}, {@link ensembles.EnsembleTriTableau2}
 * @author davidroussel
 */
@RunWith(value = Parameterized.class)
public class AllEnsembleTest
{
    /**
     * l'ensemble Ã  tester
     */
    private Ensemble<String> ensemble;

    /**
     * Le type d'ensemble Ã  tester.
     */
    private Class<? extends Ensemble<String>> typeEnsemble;

    /**
     * Nom du type d'ensemble Ã  tester
     */
    private String typeName;

    /**
     * Les diffÃ©rentes natures d'ensembles Ã  tester
     */
    @SuppressWarnings("unchecked")
    private static final Class<? extends Ensemble<String>>[] typesEnsemble =
    (Class<? extends Ensemble<String>>[]) new Class<?>[]
    {
        /*
         * TODO Commenter / dÃ©commenter les lignes ci-dessous en fonction
         * de votre avancement (Attention la derniÃ¨re ligne non commentÃ©e
         * ne doit pas avoir de virgule)
         */
        EnsembleTableau.class,
//      EnsembleVector.class,
//      EnsembleListe.class,
//      EnsembleTriVector.class,
//      EnsembleTriVector2.class,
//      EnsembleTriTableau.class
//      EnsembleTriTableau2.class,
//      EnsembleTriListe.class,
//      EnsembleTriListe2.class
    };

    /**
     * Elements pour remplir l'ensemble : "Lorem ipsum dolor sit amet"
     */
    private static final String[] elements1 = new String[] {
        "Lorem",
        "ipsum",
        "sit",
```



```java
        "dolor",
        "amet"
    };

    /**
     * Autres Elements pour remplir un ensemble :
     * "dolor amet consectetur adipisicing elit"
     */
    private static final String[] elements2 = new String[] {
        "dolor",
        "amet",
        "consectetur",
        "adipisicing",
        "elit"
    };

    /**
     * Elements union de {@value #elements1} et {@link #elements2}
     */
    private static final String[] allSingleElements = new String[] {
        "Lorem",
        "ipsum",
        "sit",
        "dolor",
        "amet",
        "consectetur",
        "adipisicing",
        "elit"
    };

    /**
     * Elements union triÃ©e de {@value #elements1} et
     * {@link #elements2}
     */
    private static final String[] allSingleElementsSorted = new String[] {
        "Lorem",
        "adipisicing",
        "amet",
        "consectetur",
        "dolor",
        "elit",
        "ipsum",
        "sit"
    };

    /**
     * Elements communs Ã  {@value #elements1} et {@link #elements2}
     */
    private static final String[] commonSingleElements = new String[] {
        "dolor",
        "amet"
    };

    /**
     * Elements du complement de {@value #elements1} et
     * {@link #elements2}
     */
    private static final String[] complementElements1 = new String[] {
        "Lorem",
        "ipsum",
        "sit"
    };

    /**
     * Elements du complement de {@value #elements2} et
     * {@link #elements1}
     */
    private static final String[] complementElements2 = new String[] {
        "consectetur",
        "adipisicing",
        "elit"
    };

    /**
     * Elements non communs Ã  {@value #elements1} et
     * {@link #elements2}
     */
    private static final String[] diffSingleElements = new String[] {
        "Lorem",
        "ipsum",
        "sit",
        "consectetur",
        "adipisicing",
        "elit"
    };

    /**
     * Elements pour remplir l'ensemble avec des doublons pour vÃ©rifier que ceux
     * ci ne seront pas ajoutÃ©s dans les ensembles
     */
```

```
181     private static final String[] elements = new String[elements1.length
182             + elements2.length];
183
184     /**
185      * Collection pour contenir les Ã©lements de remplissage
186      */
187     private ArrayList<String> listElements;
188
189     /**
190      * Construit une instance de Ensemble<String> en fonction d'un type
191      * d'ensemble Ã  crÃ©er et Ã©ventuellement d'un contenu l'ensemble Ã  mettre en
192      * place
193      *
194      * @param testName le message Ã  rÃ©pÃ©ter dans les assertions en fonction du
195      *            test dans lequel est employÃ© cette mÃ©thode
196      * @param type le type d'ensemble Ã  crÃ©er
197      * @param content le contenu Ã  mettre en place dans le nouvel ensemble, ou
198      * bien null si aucun contenu n'est requis.
199      * @return un nouvel ensemble du type demandÃ© evt rempli avec le contenu
200      * fournit s'il est non null.
201      */
202     private static Ensemble<String>
203     constructEnsemble(String testName,
204                       Class<? extends Ensemble<String>> type,
205                       Iterable<String> content)
206     {
207         Ensemble<String> ensemble = null;
208
209         try
210         {
211             ensemble = EnsembleFactory.<String>getEnsemble(type, content);
212         }
213         catch (SecurityException e)
214         {
215             fail(testName +  " constructor security exception");
216         }
217         catch (NoSuchMethodException e)
218         {
219             fail(testName +  " constructor not found");
220         }
221         catch (IllegalArgumentException e)
222         {
223             fail(testName + " wrong constructor arguments");
224         }
225         catch (InstantiationException e)
226         {
227             fail(testName + " instanciation exception");
228         }
229         catch (IllegalAccessException e)
230         {
231             fail(testName + " illegal access");
232         }
233         catch (InvocationTargetException e)
234         {
235             fail(testName + " invocation exception");
236         }
237
238         return ensemble;
239     }
240
241     /**
242      * Compare les Ã©lÃ©ments d'un ensemble pour vÃ©rifier qu'ils sont tous dans
243      * un tableau donnÃ©
244      * @param testName le nom du test dans lequel est utilisÃ© cette mÃ©thode
245      * @param ensemble l'ensemble dont on doit comparer les Ã©lÃ©ments
246      * @param array le tableau utilisÃ© pour vÃ©rifier la prÃ©sence des Ã©lÃ©ments
247      * de l'ensemble
248      * @return true si tous les Ã©lÃ©ments du tableau sont prÃ©sents dans l'ensemble
249      */
250     private static boolean compareElts2Array(String testName,
251             Ensemble<String> ensemble, String[] array)
252     {
253         for (String elt : array)
254         {
255             boolean contenu = ensemble.contient(elt);
256             assertTrue(testName + " contient(" + elt + ")failed", contenu);
257             if (¬contenu)
258             {
259                 return false;
260             }
261         }
262         return true;
263     }
264
265     /**
266      * VÃ©rifie qu'un ensemble ne contient qu'un seul exemplaire de chacun
267      * de ses Ã©lÃ©ments
268      * @param testName le nom du test dans lequel est employÃ©e cette mÃ©thode
269      * @param ensemble l'ensemble Ã  tester
270      * @return true si chaque Ã©lÃ©ment de l'ensemble n'existe qu'Ã  un seul
```



```
271      * exemplaire.
272      */
273     private static <E> boolean checkCount(String testName, Ensemble<E> ensemble)
274     {
275         Map<E, Integer> wordCount = new HashMap<E, Integer>();
276         for (E elt : ensemble)
277         {
278             if (¬wordCount.containsKey(elt))
279             {
280                 wordCount.put(elt, Integer.valueOf(1));
281             }
282             else
283             {
284                 Integer count = wordCount.get(elt);
285                 count = Integer.valueOf(count.intValue() + 1);
286                 wordCount.put(elt, count);
287             }
288         }
289
290         for (Integer i : wordCount.values())
291         {
292             int countValue = i.intValue();
293             assertEquals(testName + " count check #" + countValue + " failed",
294                     1, countValue);
295             if (countValue ≠ 1)
296             {
297                 return false;
298             }
299         }
300
301         return true;
302     }
303
304     /**
305      * MÃ©lange les Ã©lÃ©ments d'un tableau
306      * @param elements les Ã©lements Ã  mÃ©langer
307      * @return un tableau de mÃªme dimension avec les Ã©lÃ©ments dans un autre
308      * ordre
309      */
310     private static String[] shuffleElements(String[] elements)
311     {
312         List<String> listElements = Arrays.asList(elements);
313
314         Collections.shuffle(listElements);
315
316         String[] result = new String[elements.length];
317         int i = 0;
318         for (String elt : listElements)
319         {
320             result[i++] = elt;
321         }
322
323         return result;
324     }
325
326     /**
327      * ParamÃ¨tres Ã  transmettre au constructeur de la classe de test.
328      *
329      * @return une collection de tableaux d'objet contenant les paramÃ¨tres Ã 
330      *         transmettre au constructeur de la classe de test
331      */
332     @Parameters(name = "{index}:{1}")
333     public static Collection<Object[]> data()
334     {
335         Object[][] data = new Object[typesEnsemble.length][2];
336         for (int i = 0; i < typesEnsemble.length; i++)
337         {
338             data[i][0] = typesEnsemble[i];
339             data[i][1] = typesEnsemble[i].getSimpleName();
340         }
341
342         return Arrays.asList(data);
343     }
344
345     /**
346      * Constructeur paramÃªtrÃ© par le type d'ensemble Ã  tester.
347      * LancÃ© pour chaque test
348      * @param typeEnsemble le type d'ensemble Ã  qÃ©nÃ©rer
349      * @param le nom du type d'ensemble Ã  tester (pour le faire apparaÃ®tre
350      * dans le dÃ©roulement des tests).
351      */
352     public AllEnsembleTest(Class<? extends Ensemble<String>> typeEnsemble,
353                            String typeEnsembleName)
354     {
355         this.typeEnsemble = typeEnsemble;
356         typeName = typeEnsembleName;
357     }
358
359     /**
360      * Mise en place avant l'ensemble des tests
```

```
361      * @throws java.lang.Exception
362      */
363     @BeforeClass
364     public static void setUpBeforeClass() throws Exception
365     {
366         int j = 0;
367         for (int i = 0; i < elements1.length; i++)
368         {
369             elements[j++] = elements1[i];
370         }
371         for (int i = 0; i < elements2.length; i++)
372         {
373             elements[j++] = elements2[i];
374         }
375         System.out.println("------------------------------------------------");
376         System.out.println("Test des ensembles");
377         System.out.println("------------------------------------------------");
378     }
379
380     /**
381      * Nettoyage aprÃ¨s l'ensemble des tests
382      * @throws java.lang.Exception
383      */
384     @AfterClass
385     public static void tearDownAfterClass() throws Exception
386     {
387         System.out.println("------------------------------------------------");
388         System.out.println("Fin Test des ensembles");
389         System.out.println("------------------------------------------------");
390     }
391
392     /**
393      * Mise en place avant chaque test
394      * @throws java.lang.Exception
395      */
396     @Before
397     public void setUp() throws Exception
398     {
399         ensemble = constructEnsemble("setUp", typeEnsemble, null);
400         assertNotNull("setUp non null ensemble failed", ensemble);
401
402         listElements = new ArrayList<String>();
403         for (String elt : elements)
404         {
405             listElements.add(elt);
406         }
407     }
408
409     /**
410      * Nettoyage aprÃ¨s chaque test
411      * @throws java.lang.Exception
412      */
413     @After
414     public void tearDown() throws Exception
415     {
416         ensemble.efface();
417         ensemble = null;
418         listElements.clear();
419         listElements = null;
420     }
421
422     /**
423      * Test method for {@link ensembles.EnsembleVector#EnsembleVector()} or
424      * {@link ensembles.EnsembleListe#EnsembleListe()} or
425      * {@link ensembles.EnsembleTableau#EnsembleTableau()}
426      */
427     @Test
428     public final void testDefaultConstructor()
429     {
430         String testName = new String(typeName + "()");
431         System.out.println(testName);
432
433         ensemble = constructEnsemble(testName, typeEnsemble, null);
434         assertNotNull(testName + " non null instance failed", ensemble);
435
436         assertEquals(testName + " instance type failed", typeEnsemble,
437                 ensemble.getClass());
438         assertTrue(testName +  " empty instance failed", ensemble.estVide());
439         assertEquals(testName + " instance size failed", 0, ensemble.cardinal());
440     }
441
442     /**
443      * Test method for {@link ensembles.EnsembleVector#EnsembleVector(Iterable)}
444      * or {@link ensembles.EnsembleListe#EnsembleListe(Iterable)} or
445      * {@link ensembles.EnsembleTableau#EnsembleTableau(Iterable)}
446      */
447     @Test
448     public final void testCopyConstructor()
449     {
450         String testName = new String(typeName + "(Iterable)");
```



```
451         System.out.println(testName);
452
453         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
454         assertNotNull(testName + " non null instance failed", ensemble);
455
456         assertEquals(testName + " instance type failed", typeEnsemble,
457                 ensemble.getClass());
458         assertFalse(testName +  " not empty instance failed", ensemble.estVide());
459         boolean compare = compareElts2Array(testName, ensemble, allSingleElements);
460         assertTrue(testName + "elts compare failed", compare);
461
462         // Tous les Ã©lÃ©ments de ensemble doivent se retrouver dans list
463         for (String elt : ensemble)
464         {
465             assertTrue(testName + "check content [" + elt + "] failed",
466                     listElements.contains(elt));
467         }
468
469         // Tous les Ã©lÃ©ments de l'ensemble n'existent qu'Ã  un seul exemplaire
470         boolean countCheck = AllEnsembleTest.<String>checkCount(testName, ensemble);
471
472         assertTrue(testName + "after count check failed", countCheck);
473     }
474
475     /**
476      * Test method for {@link ensembles.Ensemble#ajout(java.lang.Object)}.
477      */
478     @Test
479     public final void testAjout()
480     {
481         String testName = new String(typeName + ".ajout(E)");
482         System.out.println(testName);
483
484         // Ensemble vide avant remplissage
485         assertEquals(testName + " ensemble vide failed", 0, ensemble.cardinal());
486         int count = 0;
487         for (String elt : elements)
488         {
489             if (¬ensemble.contient(elt))
490             {
491                 count++;
492             }
493             ensemble.ajout(elt);
494         }
495         // Ensemble non vide aprÃ¨s remplissage
496         assertEquals(testName + " ensemble rempli failed", count,
497                 ensemble.cardinal());
498
499         // Verif taille ensemble
500         boolean countCheck = AllEnsembleTest.<String>checkCount(testName, ensemble);
501         assertTrue(testName + "after count check failed", countCheck);
502
503         // Comparaison des elts avec allSingleElements
504         boolean compare = compareElts2Array(testName, ensemble, allSingleElements);
505         assertTrue(testName + "elts compare failed", compare);
506
507         // Ajout d'un elt null
508         boolean ajoutNull = ensemble.ajout(null);
509         assertFalse(testName + " ajout null is true", ajoutNull);
510     }
511
512     /**
513      * Test method for {@link ensembles.Ensemble#retrait(java.lang.Object)}.
514      */
515     @Test
516     public final void testRetrait()
517     {
518         String testName = new String(typeName + ".retrait(E)");
519         System.out.println(testName);
520
521         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
522         assertNotNull(testName + " non null instance failed", ensemble);
523
524         String[] elementsToRemove = shuffleElements(allSingleElements);
525
526         for (String elt : elementsToRemove)
527         {
528             ensemble.retrait(elt);
529
530             assertFalse(testName + "no more contains " + elt + " failed",
531                     ensemble.contient(elt));
532         }
533
534         assertTrue(testName + " ensemble vide aprÃ¨s retraits failed",
535                 ensemble.estVide());
536     }
537
538     /**
539      * Test method for {@link ensembles.Ensemble#estVide()}.
540      */
```

```
541     @Test
542     public final void testEstVide()
543     {
544         String testName = new String(typeName + ".estVide()");
545         System.out.println(testName);
546
547         assertTrue(testName + " ensemble vide failed", ensemble.estVide());
548         assertFalse(testName + " ens vide rien Ã  itÃ©rer failed",
549                 ensemble.iterator().hasNext());
550
551         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
552         assertNotNull(testName + " non null instance failed", ensemble);
553
554         assertFalse(testName + " ensemble vide failed", ensemble.estVide());
555         assertTrue(testName + " ens non vide iterable failed",
556                 ensemble.iterator().hasNext());
557     }
558
559     /**
560      * Test method for {@link ensembles.Ensemble#contient(java.lang.Object)}.
561      */
562     @Test
563     public final void testContientENull()
564     {
565         String testName = new String(typeName + ".contient((E)null)");
566         System.out.println(testName);
567         String mot = null;
568
569         // Contient null sur ensemble vide
570         assertFalse(testName + " ens vide !contient(null) failed",
571                 ensemble.contient(mot));
572
573         // remplissage ensemble
574         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
575         assertNotNull(testName + " non null instance failed", ensemble);
576         assertEquals(testName + " instance remplie failed",
577                 allSingleElements.length, ensemble.cardinal());
578
579         // Contient null sur ensemble non vide
580         assertFalse(testName + " ens plein !contient(null) failed",
581                 ensemble.contient((String) null));
582     }
583
584     /**
585      * Test method for {@link ensembles.Ensemble#contient(java.lang.Object)}.
586      */
587     @Test
588     public final void testContientE()
589     {
590         String testName = new String(typeName + ".contient(E)");
591         System.out.println(testName);
592         String mot = new String("Bonjour");
593
594         // Contient mot quelconque sur ensemble vide
595         assertFalse(testName + " ens vide !contient(" + mot + ") failed",
596                 ensemble.contient(mot));
597
598         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
599         assertNotNull(testName + " non null instance failed", ensemble);
600
601         // Contient mot quelconque sur ensemble non vide
602         assertFalse(testName + " ens vide contient(" + mot + ") failed",
603                 ensemble.contient(mot));
604
605         // Contient mots contenus
606         boolean compare = compareElts2Array(testName, ensemble, allSingleElements);
607         assertTrue(testName + " elts compare failed", compare);
608     }
609
610     /**
611      * Test method for {@link ensembles.Ensemble#contient(ensembles.Ensemble)}.
612      */
613     @Test
614     public final void testContientEnsembleNull()
615     {
616         String testName = new String(typeName + ".contient((Ensemble<E>)null)");
617         System.out.println(testName);
618
619         // !Contient ensemble null dans ensemble vide
620         assertFalse(testName + "ens vide !contient(null) failed",
621                 ensemble.contient((Ensemble<String>) null));
622
623         // !Contient ensemble null dans ensemble plein
624         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
625         assertNotNull(testName + " non null instance failed", ensemble);
626         assertEquals(testName + " instance remplie taille failed",
627                 allSingleElements.length, ensemble.cardinal());
628
629         assertFalse(testName + "ens plein non !contient(null) failed",
630                 ensemble.contient((Ensemble<String>) null));
```



```
631     }
632
633     /**
634      * Test method for {@link ensembles.Ensemble#contient(ensembles.Ensemble)}.
635      */
636     @Test
637     public final void testContientEnsembleOfE()
638     {
639         for (int i = 0; i < typesEnsemble.length; i++)
640         {
641             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
642             String otherTypeName = otherType.getSimpleName();
643
644             String testName = new String(typeName + ".contient("
645                     + otherTypeName + "<E>)");
646             System.out.println(testName);
647
648             // sous ensemble vide
649             Ensemble<String> sousEnsemble = constructEnsemble(testName,
650                     typesEnsemble[i], null);
651             assertNotNull(testName + " sousEnsemble non null instance failed",
652                     sousEnsemble);
653
654             // Contient sous ensemble vide dans ensemble vide
655             assertTrue(testName + " ens vide contient sous ens["
656                     + typesEnsemble[i].getSimpleName() + "] vide failed",
657                     ensemble.contient(sousEnsemble));
658
659             // remplissage ensemble
660             for (String elt : elements1)
661             {
662                 ensemble.ajout(elt);
663             }
664
665             // Contient sous ensemble vide dans ensemble non vide
666             assertTrue(testName + " ens plein contient sous ens["
667                     + typesEnsemble[i].getSimpleName() + "] vide failed",
668                     ensemble.contient(sousEnsemble));
669
670             // remplissage sous ensemble
671             for (int j = 0; j < (elements1.length / 2); j++)
672             {
673                 sousEnsemble.ajout(elements1[j]);
674             }
675
676             // Contient sous ensemble non vide ds ens non vide
677             assertTrue(testName + " ens plein contient sous ens["
678                     + typesEnsemble[i].getSimpleName() + "] failed",
679                     ensemble.contient(sousEnsemble));
680
681             // !Contient sous ensemble non vide non contenu ds ens non vide
682             sousEnsemble.ajout("consectetur");
683             assertFalse(testName + " ens plein !contient sous ens["
684                     + typesEnsemble[i].getSimpleName() + "] failed",
685                     ensemble.contient(sousEnsemble));
686
687             ensemble.efface();
688         }
689     }
690
691     /**
692      * Test method for {@link ensembles.Ensemble#efface()}.
693      */
694     @Test
695     public final void testEfface()
696     {
697         String testName = new String(typeName + ".efface()");
698         System.out.println(testName);
699
700         assertTrue(testName + "ens vide avant effacement failed",
701                 ensemble.estVide());
702
703         // Effacement ensemble vide
704         ensemble.efface();
705         assertTrue(testName + "ens vide aprÃ¨s effacement failed", ensemble.estVide());
706
707         // Effacement ensemble non vide
708         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
709         assertNotNull(testName + " non null instance failed", ensemble);
710         assertFalse(testName + "ens non vide aprÃ¨s remplissage failed",
711                 ensemble.estVide());
712         ensemble.efface();
713         assertTrue(testName + "ens vide aprÃ¨s remplissage & effacement failed",
714                 ensemble.estVide());
715     }
716
717     /**
718      * Test method for {@link ensembles.Ensemble#cardinal()}.
719      */
720     @Test
```

```
721     public final void testCardinal()
722     {
723         String testName = new String(typeName + ".cardinal()");
724         System.out.println(testName);
725 
726         assertTrue(testName + " ensemble vide failed", ensemble.estVide());
727         assertEquals(testName + " cardinal 0 sur ensemble vide failed", 0,
728                 ensemble.cardinal());
729 
730         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
731         assertNotNull(testName + " non null instance failed", ensemble);
732 
733         assertFalse(testName + " ensemble non vide failed", ensemble.estVide());
734         assertEquals(testName + " cardinal " + allSingleElements.length
735                 + " sur ensemble rempli failed", allSingleElements.length,
736                 ensemble.cardinal());
737     }
738 
739     /**
740      * Test method for {@link ensembles.Ensemble#union(ensembles.Ensemble)}.
741      */
742     @Test
743     public final void testUnion()
744     {
745         for (int i = 0; i < typesEnsemble.length; i++)
746         {
747             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
748             String otherTypeName = otherType.getSimpleName();
749 
750             String testName = new String(typeName + ".union(" + otherTypeName
751                     + "<E>)");
752             System.out.println(testName);
753 
754             // remplissage ensemble avec singleElements
755             for (String elt : elements1)
756             {
757                 ensemble.ajout(elt);
758             }
759 
760             // remplissage other avec singleElements2
761             Ensemble<String> other = constructEnsemble(testName,
762                     typesEnsemble[i], null);
763             assertNotNull(testName + " other instance non null failed", other);
764             for (String elt : elements2)
765             {
766                 other.ajout(elt);
767             }
768 
769             Ensemble<String> union = ensemble.union(other);
770 
771             assertNotNull(testName + " non null union instance failed", union);
772             assertFalse(testName + " self union", ensemble == union);
773             assertFalse(testName + " self union", other == union);
774             assertEquals(testName + " taille failed",
775                     allSingleElements.length, union.cardinal());
776             boolean compare = compareElts2Array(testName, union,
777                     allSingleElements);
778             assertTrue(testName + " elts compare failed", compare);
779         }
780     }
781 
782     /**
783      * Test method for {@link ensembles.Ensemble#intersection(ensembles.Ensemble)}.
784      */
785     @Test
786     public final void testIntersection()
787     {
788         for (int i = 0; i < typesEnsemble.length; i++)
789         {
790             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
791             String otherTypeName = otherType.getSimpleName();
792 
793             String testName = new String(typeName + ".intersection("
794                     + otherTypeName + "<E>)");
795             System.out.println(testName);
796 
797             // remplissage ensemble avec singleElements
798             for (String elt : elements1)
799             {
800                 ensemble.ajout(elt);
801             }
802 
803             // remplissage other avec singleElements2
804             Ensemble<String> other = constructEnsemble(testName,
805                     typesEnsemble[i], null);
806             assertNotNull(testName + " other non null instance failed", other);
807             for (String elt : elements2)
808             {
809                 other.ajout(elt);
810             }
```



```
811 
812             Ensemble<String> intersection = ensemble.intersection(other);
813 
814             assertNotNull(testName + " non null intersection instance failed",
815                     intersection);
816             assertFalse(testName + " self intersection", ensemble == intersection);
817             assertFalse(testName + " self intersection", other == intersection);
818             assertEquals(testName + " taille failed",
819                     commonSingleElements.length, intersection.cardinal());
820             boolean compare = compareElts2Array(testName, intersection,
821                     commonSingleElements);
822             assertTrue(testName + " elts compare failed", compare);
823         }
824     }
825 
826     /**
827      * Test method for {@link ensembles.Ensemble#complement(ensembles.Ensemble)}.
828      */
829     @Test
830     public final void testComplement()
831     {
832         for (int i = 0; i < typesEnsemble.length; i++)
833         {
834             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
835             String otherTypeName = otherType.getSimpleName();
836 
837             String testName = new String(typeName + ".complement("
838                     + otherTypeName + "<E>)");
839             System.out.println(testName);
840 
841             // remplissage ensemble avec singleElements
842             for (String elt : elements1)
843             {
844                 ensemble.ajout(elt);
845             }
846 
847             // remplissage other avec singleElements2
848             Ensemble<String> other = constructEnsemble(testName,
849                     typesEnsemble[i], null);
850             assertNotNull(testName + " other non null instance failed", other);
851             for (String elt : elements2)
852             {
853                 other.ajout(elt);
854             }
855 
856             Ensemble<String> complement1 = ensemble.complement(other);
857 
858             assertNotNull(testName + " non null complement instance 1 failed",
859                     complement1);
860             assertFalse(testName + " self complement1", ensemble == complement1);
861             assertFalse(testName + " self complement1", other == complement1);
862             assertEquals(testName + " taille 1 failed",
863                     complementElements1.length, complement1.cardinal());
864             boolean compare = compareElts2Array(testName, complement1,
865                     complementElements1);
866             assertTrue(testName + " elts compare 1 failed", compare);
867 
868             Ensemble<String> complement2 = other.complement(ensemble);
869 
870             assertNotNull(testName + " non null complement instance 2 failed",
871                     complement2);
872             assertFalse(testName + " self complement2", ensemble == complement2);
873             assertFalse(testName + " self complement2", other == complement2);
874             assertEquals(testName + " taille 2 failed",
875                     complementElements2.length, complement2.cardinal());
876             compare = compareElts2Array(testName, complement2,
877                     complementElements2);
878             assertTrue(testName + "elts compare 2 failed", compare);
879         }
880     }
881 
882     /**
883      * Test method for {@link ensembles.Ensemble#difference(ensembles.Ensemble)}.
884      */
885     @Test
886     public final void testDifference()
887     {
888         for (int i = 0; i < typesEnsemble.length; i++)
889         {
890             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
891             String otherTypeName = otherType.getSimpleName();
892 
893             String testName = new String(typeName + ".difference("
894                     + otherTypeName + "<E>)");
895             System.out.println(testName);
896 
897             // remplissage ensemble avec singleElements
898             for (String elt : elements1)
899             {
900                 ensemble.ajout(elt);
```

```
901             }
902
903             // remplissage other avec singleElements2
904             Ensemble<String> other = constructEnsemble(testName,
905                     typesEnsemble[i], null);
906             assertNotNull(testName + " other non null instance failed", other);
907
908             for (String elt : elements2)
909             {
910                 other.ajout(elt);
911             }
912
913             Ensemble<String> difference = ensemble.difference(other);
914
915             assertNotNull(testName + " difference non null instance failed",
916                     difference);
917             assertFalse(testName + " self difference", ensemble ≡ difference);
918             assertFalse(testName + " self difference", other ≡ difference);
919             assertEquals(testName + " taille failed", diffSingleElements.length,
920                     difference.cardinal());
921             boolean compare = compareElts2Array(testName, difference,
922                     diffSingleElements);
923             assertTrue(testName + " elts compare failed", compare);
924         }
925     }
926
927     /**
928      * Test method for {@link ensembles.Ensemble#typeElements()}.
929      */
930     @Test
931     public final void testTypeElements()
932     {
933         String testName = new String(typeName + ".typeElements()");
934         System.out.println(testName);
935
936         assertNotNull(testName + " non null instance failed", ensemble);
937
938         // type elt sur ensemble vide == null
939         assertEquals(testName + " sur ens vide failed", null,
940                 ensemble.typeElements());
941
942         // type elt sur ensemble non vide == String
943         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
944         assertNotNull(testName + " non null instance failed", ensemble);
945         assertEquals(testName + " sur ens non vide failed", String.class,
946                 ensemble.typeElements());
947     }
948
949     /**
950      * Test method for {@link ensembles.Ensemble#equals(java.lang.Object)}.
951      */
952     @Test
953     public final void testEquals()
954     {
955         String testName = new String(typeName + ".equals(Object)");
956         System.out.println(testName);
957
958         // Equals sur null
959         assertFalse(testName + " sur null failed", ensemble.equals(null));
960
961         // Equals sur this
962         assertTrue(testName + " sur this failed", ensemble.equals(ensemble));
963
964         // Equals sur autre objet
965         assertFalse(testName + " sur Object failed",
966                 ensemble.equals(new Object()));
967
968         // remplissage ensemble
969         for (String elt : allSingleElementsSorted)
970         {
971             ensemble.ajout(elt);
972         }
973
974         String[] allsingleElementsShuffle = shuffleElements(allSingleElements);
975
976         for (int i = 0; i < typesEnsemble.length; i++)
977         {
978             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
979             String otherTypeName = otherType.getSimpleName();
980
981             Ensemble<String> other = constructEnsemble(testName,
982                     typesEnsemble[i], null);
983
984             // Equals sur Ensemble mÃªme contenu mÃªme ordre
985             assertNotNull(testName + " other non null instance failed", other);
986             for(String elt : allSingleElementsSorted)
987             {
988                 other.ajout(elt);
989             }
990             assertEquals(testName + " ens identique, ordre identique["
```



```
991                     + otherTypeName + "] failed", ensemble, other);
992
993             // Equals sur Ensemble mÃªme contenu ordre diffÃ©rent
994             other.efface();
995             for(String elt : allsingleElementsShuffle)
996             {
997                 other.ajout(elt);
998             }
999
1000            // ensemble est toujours sorted car construit avec
1001            // allSingleElementsSorted
1002            if ((ensemble instanceof EnsembleTri<?>) ∧
1003               ¬(other instanceof EnsembleTri<?>))
1004            {
1005                assertFalse(testName + " ens identique, ordre diffÃ©rent["
1006                        + otherTypeName + "] failed", ensemble.equals(other));
1007            }
1008            else
1009            {
1010                assertEquals(testName + " ens identique, ordre diffÃ©rent["
1011                        + otherTypeName + "] failed", ensemble, other);
1012            }
1013
1014            // Equals sur Ensemble contenu diffÃ©rent
1015            other.ajout("bonjour");
1016            assertFalse(testName + " ens diffÃ©rent failed",
1017                    ensemble.equals(other));
1018        }
1019    }
1020
1021    /**
1022     * Test method for {@link ensembles.Ensemble#hashCode()}.
1023     */
1024    @Test
1025    public final void testHashCode()
1026    {
1027        String testName = new String(typeName + ".hashCode()");
1028        System.out.println(testName);
1029        int hash;
1030        boolean trie = ensemble instanceof EnsembleTri<?>;
1031        if (trie)
1032        {
1033            hash = 1;
1034        }
1035        else
1036        {
1037            hash = 0;
1038        }
1039
1040        // hash code ensemble vide ==
1041        //  0 pour les Ensemble
1042        //  1 pour les EnsembleTri
1043        assertEquals(testName + " hashcode ens vide failed", hash,
1044                ensemble.hashCode());
1045
1046        // hash code ensemble non vide ==
1047        //  somme des hashcode des elts pour les Ensemble
1048        //  comme les collections pour les EnsembleTri
1049        for (String elt : allSingleElements)
1050        {
1051            ensemble.ajout(elt);
1052        }
1053        if (trie)
1054        {
1055            final int prime = 31;
1056            for (String elt : allSingleElementsSorted)
1057            {
1058                hash = (prime * hash) + (elt ≡ null ? 0 : elt.hashCode());
1059            }
1060        }
1061        else
1062        {
1063            for (String elt : allSingleElements)
1064            {
1065                hash += elt.hashCode();
1066            }
1067        }
1068
1069        assertEquals(testName + " hashcode ens non vide failed", hash,
1070                ensemble.hashCode());
1071    }
1072
1073    /**
1074     * Test method for {@link ensembles.Ensemble#toString()}.
1075     */
1076    @Test
1077    public final void testToString()
1078    {
1079        String testName = new String(typeName + ".toString()");
1080        System.out.println(testName);
```

```
1081
1082            ensemble = constructEnsemble(testName, typeEnsemble, listElements);
1083            assertNotNull(testName + " non null instance failed", ensemble);
1084
1085            StringBuilder sb = new StringBuilder();
1086            sb.append("[");
1087            Iterator<String> it = ensemble.iterator();
1088            if (it ≠ null)
1089            {
1090                for (; it.hasNext();)
1091                {
1092                    sb.append(it.next().toString());
1093                    if (it.hasNext())
1094                    {
1095                        sb.append(",");
1096                    }
1097                }
1098                sb.append("]");
1099
1100                String expected = sb.toString();
1101
1102                assertEquals(testName, expected, ensemble.toString());
1103            }
1104            else
1105            {
1106                fail(testName + " null iterator");
1107            }
1108        }
1109
1110        /**
1111         * Test method for {@link ensembles.Ensemble#iterator()}.
1112         */
1113        @Test
1114        public final void testIterator()
1115        {
1116            String testName = new String(typeName + ".iterator()");
1117            System.out.println(testName);
1118
1119            Iterator<String> it = null;
1120
1121            // iterator existe
1122            it = ensemble.iterator();
1123            assertNotNull(testName +   " non null instance failed", it);
1124
1125            // iterator sur ens vide n'a pas d'elts Ã  itÃ©rer
1126            assertFalse(testName + " !hasNext() sur ens vide failed", it.hasNext());
1127
1128            // remplissage
1129            for (String elt : allSingleElements)
1130            {
1131                ensemble.ajout(elt);
1132            }
1133
1134            it = ensemble.iterator();
1135
1136            // iterator sur ens rempli
1137            assertTrue(testName + " hashNext() sur ens rempli failed", it.hasNext());
1138
1139            String[] array;
1140            if (ensemble instanceof EnsembleTri<?>)
1141            {
1142                array = allSingleElementsSorted;
1143            }
1144            else
1145            {
1146                array = allSingleElements;
1147            }
1148
1149            // comparaison des elts
1150            for(int i = 0; (i < array.length) ∧ it.hasNext(); i++)
1151            {
1152                assertEquals(testName + "check elt:" + array[i] + " failed",
1153                        array[i], it.next());
1154            }
1155
1156            // plus l'elts Ã  itÃ©rer
1157            assertFalse(testName + " !hasNext() fin comparaison failed",
1158                    it.hasNext());
1159
1160            // retrait des elts avec l'itÃ©rateur
1161            it = ensemble.iterator();
1162            for (int i = 0; (i < array.length) ∧ it.hasNext(); i++)
1163            {
1164                it.next();
1165                it.remove();
1166                assertFalse(testName + " retrait elt:" + array[i] + " failed",
1167                        ensemble.contient(array[i]));
1168            }
1169
1170            // plus l'elts Ã  itÃ©rer
```



```
1171            assertFalse(testName + " !hasNext() fin retrait failed", it.hasNext());
1172            assertTrue(testName + " ens vide aprÃ¨s retraits failed",
1173                    ensemble.estVide());
1174        }
1175    }
```

```java
package tests;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertNotSame;
import static org.junit.Assert.assertSame;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.NoSuchElementException;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import listes.Liste;

/**
 * Classe de test de la liste ChainÃ©e
 * @author davidroussel
 */
public class ListeTest
{

    /**
     * La liste Ã  tester.
     * La nature du contenu de la liste importe peu du moment qu'il est
     * homogÃ¨ne : donc n'importe quel type ferait l'affaire.
     */
    private Liste<String> liste = null;

    /**
     * Liste des Ã©lÃ©ments Ã  insÃ©rer dans la liste
     */
    private static String[] elements;

    /**
     * Mise en place avant l'ensemble des tests
     * @throws java.lang.Exception
     */
    @BeforeClass
    public static void setUpBeforeClass() throws Exception
    {
        System.out.println("----------------------------------------------------");
        System.out.println("Test de la Liste");
        System.out.println("----------------------------------------------------");
    }

    /**
     * Nettoyage aprÃ¨s l'ensemble des tests
     * @throws java.lang.Exception
     */
    @AfterClass
    public static void tearDownAfterClass() throws Exception
    {
        System.out.println("----------------------------------------------------");
        System.out.println("Fin Test de la Liste");
        System.out.println("----------------------------------------------------");
    }

    /**
     * Mise en place avant chaque test
     * @throws java.lang.Exception
     */
    @Before
    public void setUp() throws Exception
    {
        elements = new String[] {
                    "Hello",
                    "Brave",
                    "New",
                    "World"
                };
        liste = new Liste<String>();
    }

    /**
     * Nettoyage aprÃ¨s chaque test
     * @throws java.lang.Exception
     */
    @After
    public void tearDown() throws Exception
    {
        liste.clear();
        liste = null;
```



```java
    }

    /**
     * MÃ©thode utilitaire de remplissage de la liste avec les Ã©lÃ©ments
     * du tableau #elements
     */
    private final void remplissage()
    {
        if (liste ≠ null)
        {
            for (String elt : elements)
            {
                liste.add(elt);
            }
        }
    }

    /**
     * Test method for {@link listes.Liste#Liste()}.
     */
    @Test
    public final void testListe()
    {
        String testName = new String("Liste<String>()");
        System.out.println(testName);

        assertNotNull(testName + " instance non null failed", liste);
        assertTrue(testName + " liste vide failed", liste.empty());
    }

    /**
     * Test method for {@link listes.Liste#Liste(listes.Liste)}.
     */
    @Test
    public final void testListeListeOfT()
    {
        String testName = new String("Liste<String>(Liste<String>)");
        System.out.println(testName);

        Liste<String> liste2 = new Liste<String>();
        liste = new Liste<String>(liste2);

        assertNotNull(testName + " instance non null failed", liste);
        assertTrue(testName + " liste vide failed", liste.empty());

        remplissage();
        assertFalse(testName + " liste remplie failed", liste.empty());
        liste2 = new Liste<String>(liste);
        assertNotNull(testName + " copie liste remplie failed", liste2);
        assertEquals(testName + " contenus Ã©gaux failed", liste, liste2);
    }

    /**
     * Test method for {@link listes.Liste#add(java.lang.Object)}.
     */
    @Test
    public final void testAdd()
    {
        String testName = new String("Liste<String>.add(E)");
        System.out.println(testName);

        // Ajout dans une liste vide
        liste.add(elements[0]);
        assertFalse(testName + "liste non vide failed", liste.empty());
        Iterator<String> it = liste.iterator();
        String insertedElt = it.next();
        assertSame(testName + " contrÃ´le ref element[0] failed", insertedElt, elements[0]);
        // Si assertSame rÃ©ussit asserEquals n'est plus nÃ©cessaire

        // Ajout dans une liste non vide
        for (int i=1; i < elements.length; i++)
        {
            liste.add(elements[i]);
            /*
             * Attention le prÃ©cÃ©dent "it" a Ã©tÃ© invalidÃ© par l'ajout
             * Lors du dernier next le current de l'itÃ©rateur est passÃ© Ã  null
             * puisqu'il n'y avait pas (encore) de suivant, donc retenter un
             * next sur le mÃªme itÃ©rateur gÃ©nÃ¨rera un NoSuchelementException.
             * Il faut donc rÃ©obtenir un itÃ©rateur pour parcourir la liste
             * aprÃ¨s un ajout
             */
            it = liste.iterator();
            for (int j = 0; j ≤ i; j++)
            {
                insertedElt = it.next();
            }
            assertSame(testName + " contrÃ´le ref element[" + i + "] failed",
                    insertedElt, elements[i]);
        }
    }
```

```
181
182     /**
183      * Test method for {@link listes.Liste#add(java.lang.Object)}.
184      */
185     @Test(expected = NullPointerException.class)
186     public final void testAddNull()
187     {
188         String testName = new String("Liste<String>.add(null)");
189         System.out.println(testName);
190
191         liste.add(elements[0]);
192
193         assertFalse(testName + " ajout 1 elt failed", liste.empty());
194
195         // Ajout null dans une liste non vide (sinon on fait un insere(null))
196         // Doit lever une NullPointerException
197         liste.add(null);
198
199         fail(testName + " ajout null sans exception");
200     }
201
202     /**
203      * Test method for {@link listes.Liste#insert(java.lang.Object)}.
204      */
205     @Test
206     public final void testInsert()
207     {
208         String testName = new String("Liste<String>.insert(E)");
209         System.out.println(testName);
210
211         // Insertion elt null
212         try
213         {
214             liste.insert(null);
215
216             fail(testName +" insertion elt null");
217         }
218         catch (NullPointerException e)
219         {
220             assertTrue(testName + " insertion elt null, liste vide failed",
221                     liste.empty());
222         }
223
224         // Insertion dans une liste vide
225         int lastIndex = elements.length - 1;
226         liste.insert(elements[lastIndex]);
227         assertFalse(testName + "liste non vide failed", liste.empty());
228         Iterator<String> it = liste.iterator();
229         String insertedElt = it.next();
230         assertSame(testName + " contrÃ´le ref element[" + lastIndex + "] failed",
231                 insertedElt, elements[lastIndex]);
232         // Si assertSame rÃ©ussit asserEquals n'est plus nÃ©cessaire
233
234         // Ajout dans une liste non vide
235         for (int i=1; i < elements.length; i++)
236         {
237             liste.insert(elements[lastIndex - i]);
238
239             insertedElt = liste.iterator().next();
240             assertSame(testName + " contrÃ´le ref element[" + (lastIndex - i)
241                     + "] failed", insertedElt, elements[lastIndex - i]);
242         }
243     }
244
245     /**
246      * Test method for {@link listes.Liste#insert(java.lang.Object)}.
247      */
248     @Test(expected = NullPointerException.class)
249     public final void testInsertNull()
250     {
251         String testName = new String("Liste<String>.insert(null)");
252         System.out.println(testName);
253
254         // Insertion dans une liste vide
255         // Doit soulever une NullPointerException
256         liste.insert(null);
257
258         fail(testName + " insertion null sans exception");
259     }
260
261     /**
262      * Test method for {@link listes.Liste#insert(java.lang.Object, int)}.
263      */
264     @Test
265     public final void testInsertInt()
266     {
267         String testName = new String("Liste<String>.insert(E, int)");
268         System.out.println(testName);
269
270         int[] nextIndex = new int[] {1, 0, 3, 2};
```



```
271         int index = 0;
272
273         // - insertion d'un Ã©lÃ©ment null
274         boolean result = liste.insert(null, 0);
275         assertFalse(testName + " insertion elt null ds liste vide failed",
276                 result);
277         assertTrue(testName + " insertion elt null ds liste vide, liste vide failed",
278                 liste.empty());
279
280         // - insertion dans une liste vide avec un index invalide
281         result = liste.insert(elements[nextIndex[index]], 1);
282         assertFalse(testName + " insertion ds liste vide, index invalide failed",
283                 result);
284         assertTrue(testName + " insertion ds liste vide, index invalide, " +
285                 "liste vide failed",liste.empty());
286
287         // + insertion dans une liste vide avec un index valide
288         result = liste.insert(elements[nextIndex[index]], 0);
289         // liste = Brave ->
290         assertTrue(testName + " insertion ds liste vide, index valide failed",
291                 result);
292         assertFalse(testName + " insertion ds liste vide, index valide, " +
293                 "liste non vide failed",liste.empty());
294         index++;
295
296         // - insertion dans une liste non vide avec un index invalide
297         result = liste.insert(elements[nextIndex[index]], 5);
298         assertFalse(testName + " insertion ds liste non vide, index invalide failed",
299                 result);
300
301         // + insertion en dÃ©but de liste non vide avec un index valide
302         result = liste.insert(elements[nextIndex[index]], 0);
303         // liste = Hello -> Brave ->
304         assertTrue(testName + " insertion dÃ©but liste non vide, index valide failed",
305                 result);
306         index++;
307
308         // + insertion en fin de liste non vide avec un index valide
309         result = liste.insert(elements[nextIndex[index]], 2);
310         // liste = Hello -> Brave -> World
311         assertTrue(testName + " insertion fin liste non vide, index valide failed",
312                 result);
313         index++;
314
315         // + insertion en milieu de liste non vide avec un index valide
316         result = liste.insert(elements[nextIndex[index]], 2);
317         // liste = Hello -> Brave -> New -> World
318         assertTrue(testName + " insertion milieu liste non vide, index valide failed",
319                 result);
320     }
321
322     /**
323      * Test method for {@link listes.Liste#remove(java.lang.Object)}.
324      */
325     @Test
326     public final void testRemove()
327     {
328         String testName = new String("Liste<String>.remove(E)");
329         System.out.println(testName);
330
331         // suppression d'un Ã©lÃ©ment non null d'une liste vide
332         boolean result = liste.remove(elements[0]);
333         assertTrue(testName + " elt liste vide failed", liste.empty());
334         assertFalse(testName + " elt liste vide failed", result);
335
336         // suppresion d'un Ã©lement null d'une liste vide
337         result = liste.remove(null);
338         assertTrue(testName + " null liste vide failed", liste.empty());
339         assertFalse(testName + " null liste vide failed", result);
340
341         remplissage();
342         liste.add("Hello"); // "Hello" not same as elements[0]
343         // liste = Hello -> Brave -> New -> World -> Hello
344
345         // suppression d'un Ã©lÃ©ment null d'une liste non vide
346         result = liste.remove(null);
347         assertFalse(testName + " null failed", result);
348
349         // suppression d'un Ã©lÃ©ment inexistant d'une liste non vide
350         result = liste.remove("Coucou");
351         assertFalse(testName + " Coucou failed", result);
352
353         // suppression d'un Ã©lement existant en dÃ©but de liste
354         result = liste.remove("Hello");
355         // liste = Brave -> New -> World -> Hello
356         assertTrue(testName + " suppr Hello debut failed", result);
357         String nextElt = liste.iterator().next();
358         assertSame(testName + " suppr Hello debut failed", nextElt, elements[1]);
359
360         // suppresion d'un Ã©lement existant en fin de liste
```

```
361         result = liste.remove("Hello");
362         // liste = Brave -> New -> World
363         assertTrue(testName + " Hello fin failed", result);
364         Iterator<String> it = liste.iterator();
365         it.next(); // Brave
366         it.next(); // New
367         String lastElt = it.next(); // World
368         assertSame(testName + " Hello fin failed", lastElt, elements[3]);
369
370         // suppression d'un Ã©lement existant en milieu de liste
371         result = liste.remove(elements[2]);
372         // liste = Brave -> World
373         assertTrue(testName + " New milieu failed", result);
374         it = liste.iterator();
375         String firstElt = it.next(); // Brave
376         lastElt = it.next(); // World
377         assertSame(testName + " first elt left failed", firstElt, elements[1]);
378         assertSame(testName + " last elt left failed", lastElt, elements[3]);
379     }
380
381     /**
382      * Test method for {@link listes.Liste#removeAll(java.lang.Object)}.
383      */
384     @Test
385     public final void testRemoveAll()
386     {
387         String testName = new String("Liste<String>.removeAll(E)");
388         System.out.println(testName);
389
390         // suppression d'un Ã©lÃ©ment non null d'une liste vide
391         boolean result = liste.removeAll(elements[0]);
392         assertTrue(testName + " supprTous elt liste vide failed", liste.empty());
393         assertFalse(testName + " supprTous elt liste vide failed", result);
394
395         // suppresion d'un Ã©lement null d'une liste vide
396         result = liste.removeAll(null);
397         assertTrue(testName + " supprTous elt null liste vide failed", liste.empty());
398         assertFalse(testName + " supprTous elt null liste vide failed", result);
399
400         elements[2] = new String("Hello");
401         remplissage();
402         liste.add("Hello"); // "Hello" not same as elements[0]
403         // liste = Hello -> Brave -> Hello -> World -> Hello
404
405         // suppression d'un Ã©lÃ©ment null d'une liste non vide
406         result = liste.removeAll(null);
407         assertFalse(testName + " supprTous elt null liste failed", result);
408
409         // suppression d'un element existant au dÃ©but, au milieu et Ã  la fin
410         result = liste.removeAll("Hello");
411         // liste = Brave -> World
412         assertTrue(testName + " supprimeTous Hello", result);
413         Iterator<String> it = liste.iterator();
414         String firstElt = it.next();
415         String lastElt = it.next();
416         assertFalse(testName + " 2 elts left failed", it.hasNext());
417         assertSame(testName + " first elt left failed", firstElt, elements[1]);
418         assertSame(testName + " last elt left failed", lastElt, elements[3]);
419     }
420
421     /**
422      * Test method for {@link listes.Liste#size()}.
423      */
424     @Test
425     public final void testSize()
426     {
427         String testName = new String("Liste<String>.size()");
428         System.out.println(testName);
429
430         // taille d'une liste vide
431         assertTrue(testName + " taille liste vide failed", liste.size() == 0);
432
433         remplissage();
434         assertFalse(testName + " remplissage failed", liste.empty());
435
436         // taille d'une liste non vide
437         assertTrue(testName + " taille liste pleine failed",
438                    liste.size() == elements.length);
439     }
440
441 //  /**
442 //   * Test method for {@link listes.Liste#get(int)}.
443 //   */
444 //  @Test
445 //  public final void testGet()
446 //  {
447 //      String testName = new String("Liste<String>.get(int)");
448 //      System.out.println(testName);
449 //
450 //      // get sur une liste vide
```



```
451 //      assertTrue(testName + " get liste vide failed", liste.get(0) == null);
452 //      assertTrue(testName + " get liste vide failed", liste.get(-1) == null);
453 //
454 //      remplissage();
455 //      assertFalse(testName + " remplissage failed", liste.empty());
456 //
457 //      // get dans une liste non vide
458 //      for (int i = -1; i <= liste.size(); i++)
459 //      {
460 //          if ((i >= 0) && (i < liste.size()))
461 //          {
462 //              assertNotNull(testName + " get(" + i + ") liste pleine failed",
463 //                      liste.get(i));
464 //              assertTrue(testName + " get(" + i + ") liste pleine failed",
465 //                      liste.get(i).equals(elements[i]));
466 //          }
467 //          else
468 //          {
469 //              assertTrue(testName + " get(" + i + ") liste pleine failed",
470 //                      liste.get(i) == null);
471 //          }
472 //      }
473 //  }
474
475     /**
476      * Test method for {@link listes.Liste#clear()}.
477      */
478     @Test
479     public final void testClear()
480     {
481         String testName = new String("Liste<String>.clear()");
482         System.out.println(testName);
483
484         // effacement d'une liste vide
485         liste.clear();
486         assertTrue(testName + " effacement liste vide failed", liste.empty());
487
488         remplissage();
489         assertFalse(testName + " remplissage failed", liste.empty());
490
491         // effacement d'une liste non vide
492         liste.clear();
493         assertTrue(testName + " effacement failed", liste.empty());
494     }
495
496     /**
497      * Test method for {@link listes.Liste#empty()}.
498      */
499     @Test
500     public final void testEmpty()
501     {
502         String testName = new String("Liste<String>.empty()");
503         System.out.println(testName);
504
505         assertTrue(testName + " vide failed", liste.empty());
506
507         remplissage();
508
509         assertFalse(testName + " non vide failed", liste.empty());
510     }
511
512     /**
513      * Test method for {@link listes.Liste#equals(java.lang.Object)}.
514      */
515     @Test
516     public final void testEqualsObject()
517     {
518         String testName = new String("Liste<String>.equals(Object)");
519         System.out.println(testName);
520
521         remplissage();
522
523         // Inegalite sur objet null
524         boolean result  = liste.equals(null);
525         assertFalse(testName + " null object failed", result);
526
527         // Egalite sur soi-mÃªme
528         result = liste.equals(liste);
529         assertTrue(testName + " self failed", result);
530
531         // Egalite sur liste copiÃ©e
532         Liste<String> liste2 = new Liste<String>(liste);
533         result = liste.equals(liste2);
534         assertTrue(testName + " copy failed", result);
535
536         // InegalitÃ© sur listes de tailles diffÃ©rentes
537         liste2.add("of Pain");
538         result = liste.equals(liste2);
539         assertFalse(testName + " copy + of Pain failed", result);
540
```

```
541         // Inegalite sur liste Ã  contenu dans une autre ordre
542         liste2.clear();
543         for (String elt : elements)
544         {
545             liste2.insert(elt);
546         }
547         result = liste.equals(liste2);
548         assertFalse(testName + " reversed copy failed", result);
549 
550         // Egalite avec une collection standard de mÃªme contenu
551         // SSI equals compare un Iterable plutÃ´t qu'une Liste
552 //      ArrayList<String> alist = new ArrayList<String>();
553 //      for (String elt : elements)
554 //      {
555 //          alist.add(elt);
556 //      }
557 //      assertTrue(testName + " equality with std Iterable failed",
558 //              liste.equals(alist));
559     }
560 
561     /**
562      * Test method for {@link listes.Liste#toString()}.
563      */
564     @Test
565     public final void testToString()
566     {
567         String testName = new String("Liste<String>.toString()");
568         System.out.println(testName);
569 
570         remplissage();
571 
572         assertEquals(testName, "[Hello->Brave->New->World]", liste.toString());
573     }
574 
575     /**
576      * Test method for {@link listes.Liste#iterator()}.
577      */
578     @Test(expected = NoSuchElementException.class)
579     public final void testIterator()
580     {
581         String testName = new String("Liste<String>.iterator()");
582         System.out.println(testName);
583 
584         Iterator<String> it = liste.iterator();
585         assertFalse(testName + " liste vide", it.hasNext());
586 
587         remplissage();
588 
589         it = liste.iterator();
590         assertTrue(testName + " liste non vide", it.hasNext());
591 
592         int i = 0;
593         while (it.hasNext())
594         {
595             String nextElt = it.next();
596             assertNotNull(testName + "next elt not null", nextElt);
597             assertSame(testName + "next elt", elements[i++], nextElt);
598             it.remove(); // ne doit pas invalider l'itÃ©rateur
599         }
600 
601         assertFalse(testName + " finished", it.hasNext());
602 
603         // Un appel supplÃ©mentaire Ã  next sur un itÃ©rateur terminÃ©
604         // doit soulever une NoSuchElementException
605         it.next();
606 
607         fail(testName + " next sur itÃ©rateur terminÃ©");
608     }
609 
610     /**
611      * Test method for {@link listes.Liste#hashCode()}.
612      */
613     @Test
614     public final void testHashCode()
615     {
616         String testName = new String("Liste<String>.hashCode()");
617         System.out.println(testName);
618 
619         // hashcode d'une liste vide = 1
620         int listeHash = liste.hashCode();
621         assertEquals(testName + " liste vide failed", 1, listeHash, 0);
622 
623         remplissage();
624 
625         // hashcode de la liste standard
626         listeHash = liste.hashCode();
627         assertEquals(testName + " liste standard failed", 1161611233, listeHash);
628 
629         /*
630          * Contrat hashCode : Si a.equals(b) alors a.hashcode() == b.hashcode()
```



```
631          */
632         Liste<String> liste2 = new Liste<String>(liste);
633         assertNotSame(testName + " egalite liste distinctes failed", liste, liste2);
634         assertEquals(testName + " egalite liste equals failed", liste, liste2);
635         assertEquals(testName + " egalite liste hashCode failed", liste.hashCode(),
636                 liste2.hashCode(), 0);
637 
638         liste2.add("Hourra");
639         assertFalse(testName + " inegalite liste equals failed", liste.equals(liste2));
640         assertFalse(testName + " inegalite liste hashCode failed",
641                 liste.hashCode() == liste2.hashCode());
642 
643         // HashCode similaire Ã  celui d'une collection standard
644         ArrayList<String> collection = new ArrayList<String>();
645         for (String elt : elements)
646         {
647             collection.add(elt);
648         }
649         int collectionHash = collection.hashCode();
650         assertEquals(testName + " hashCode standard failed", listeHash, collectionHash);
651     }
652 }
```

```
1  package tests;
2
3  import static org.junit.Assert.assertEquals;
4  import static org.junit.Assert.assertFalse;
5  import static org.junit.Assert.assertNotNull;
6  import static org.junit.Assert.assertTrue;
7  import static org.junit.Assert.fail;
8
9  import java.util.ArrayList;
10 import java.util.Collections;
11 import java.util.Iterator;
12
13 import org.junit.After;
14 import org.junit.AfterClass;
15 import org.junit.Before;
16 import org.junit.BeforeClass;
17 import org.junit.Test;
18
19 import tableaux.Tableau;
20
21 /**
22  * Classe de teste de la classe {@link tableaux.Iterable}
23  * @author davidroussel
24  */
25 public class TableauTest
26 {
27
28     /**
29      * Le tableau Ã  tester
30      */
31     private Tableau<String> tableau;
32
33     /**
34      * Des Ã©lÃ©ments pour remplir le tableau.
35      * Le nombre d'Ã©lÃ©ments doit Ãªtre supÃ©rieur Ã  {@link Iterable#INCREMENT}
36      */
37     private final static String[] elementsArray = new String[] {
38         "Hello",
39         "Brave",
40         "New",
41         "World",
42         "of",
43         "Pain"
44     };
45
46     /**
47      * Une collection standard pour comparer avec le tableau
48      */
49     private ArrayList<String> elementsCollection;
50
51     /**
52      * Mise en place avant l'ensemble des tests
53      * @throws java.lang.Exception
54      */
55     @BeforeClass
56     public static void setUpBeforeClass() throws Exception
57     {
58         System.out.println("-------------------------------------------------");
59         System.out.println("Test du Tableau");
60         System.out.println("-------------------------------------------------");
61     }
62
63     /**
64      * Netoyage aprÃ¨s l'ensmble des tests
65      * @throws java.lang.Exception
66      */
67     @AfterClass
68     public static void tearDownAfterClass() throws Exception
69     {
70         System.out.println("-------------------------------------------------");
71         System.out.println("Fin Test du Tableau");
72         System.out.println("-------------------------------------------------");
73     }
74
75     /**
76      * Mise en place avant chaque test
77      * @throws java.lang.Exception
78      */
79     @Before
80     public void setUp() throws Exception
81     {
82         tableau = new Tableau<String>();
83         elementsCollection = new ArrayList<String>();
84         for (String elt : elementsArray)
85         {
86             elementsCollection.add(elt);
87         }
88     }
89
90     /**
```



```
91      * Nettoyage aprÃ¨s chaque test
92      * @throws java.lang.Exception
93      */
94     @After
95     public void tearDown() throws Exception
96     {
97         tableau.efface();
98         tableau = null;
99         elementsCollection.clear();
100    }
101
102    /**
103     * Comparaison des Ã©lements de deux Iterables
104     * @param testName le nom du test dans lequel est appelÃ©e cette mÃ©thode
105     * @param i1 le premier iterable Ã  tester
106     * @param i2 le second iterable avec lequel comparer
107     * @return true si les deux iterables possÃ¨dent le mÃªme nombre
108     * d'Ã©lÃ©ments et que tous les Ã©lÃ©ments sont identiques et dans le mÃªme ordre
109     */
110    private boolean compareElements(String testName,
111            Iterable<String> i1,
112            Iterable<String> i2)
113    {
114        Iterator<String> it1 = i1.iterator();
115        Iterator<String> it2 = i2.iterator();
116
117        for (; it1.hasNext() ∧ it2.hasNext();)
118        {
119            String s1 = it1.next();
120            String s2 = it2.next();
121
122            assertEquals(testName + "compare " + s1 + " with " + s2, s1, s2);
123
124            if (¬s1.equals(s2))
125            {
126                return false;
127            }
128        }
129
130        return ¬it1.hasNext() ∧ ¬it2.hasNext();
131
132    }
133
134    /**
135     * Liste mÃ©langÃ©e d'index compris entre 0 et nbElements - 1;
136     *
137     * @param nbElements le nombre d'indexs
138     * @return un tableau contenant nbElements Ã©lements compris entre
139     *         [0..nbElements-1] et mÃ©langÃ©s dans un ordre alÃ©aloire
140     */
141    private int[] shuffledIndexs(int nbElements)
142    {
143        int[] shuffled = new int[nbElements];
144
145        ArrayList<Integer> list = new ArrayList<Integer>();
146        for (int i = 0; i < nbElements; i++)
147        {
148            list.add(Integer.valueOf(i));
149        }
150
151        Collections.shuffle(list);
152
153        Iterator<Integer> il = list.iterator();
154        for (int i = 0; (i < nbElements) ∧ il.hasNext(); i++)
155        {
156            shuffled[i] = il.next().intValue();
157        }
158
159        return shuffled;
160
161    }
162
163    /**
164     * Test method for {@link tableaux.Iterable#Iterable()}.
165     */
166    @Test
167    public final void testTableau()
168    {
169        String testName = new String("Tableau()");
170        System.out.println(testName);
171
172        assertNotNull(testName + " instance", tableau);
173        assertEquals(testName + " tableau vide", tableau.taille(), 0);
174    }
175
176    /**
177     * Test method for {@link tableaux.Iterable#Iterable(java.lang.Iterable)}.
178     */
179    @Test
180    public final void testTableauIterableOfE()
```

```
181     {
182         String testName = new String("Tableau(Iterable<E>)");
183         System.out.println(testName);
184
185         tableau = new Tableau<String>(elementsCollection);
186
187         assertNotNull(testName + " instance", tableau);
188         assertEquals(testName + " tableau non vide", tableau.taille(),
189                 elementsCollection.size());
190
191         boolean compare = compareElements(testName, tableau, elementsCollection);
192
193         assertTrue(testName + " elements comparison result", compare);
194     }
195
196     /**
197      * Test method for {@link tableaux.Iterable#taille()}.
198      */
199     @Test
200     public final void testTaille()
201     {
202         String testName = new String("Tableau.taille()");
203         System.out.println(testName);
204
205         assertEquals(testName + " tableau vide", tableau.taille(), 0);
206         int taille = 0;
207         for (String elt : elementsArray)
208         {
209             tableau.ajouter(elt);
210             taille++;
211             assertEquals(testName + " tableau[" + taille + "]",
212                     tableau.taille(), taille);
213         }
214
215         tableau.efface();
216         assertEquals(testName + " tableau nettoyÃ©", tableau.taille(), 0);
217     }
218
219     /**
220      * Test method for {@link tableaux.Iterable#capacite()}.
221      */
222     @Test
223     public final void testCapacite()
224     {
225         String testName = new String("Tableau.capacite()");
226         System.out.println(testName);
227         int predictedCapacity = 0;
228
229         assertEquals(testName + "capacite tableau vide", tableau.capacite(),
230                 predictedCapacity);
231
232         int nb = 0;
233         for (String elt : elementsArray)
234         {
235             nb++;
236             if (nb > tableau.capacite())
237             {
238                 predictedCapacity += Tableau.INCREMENT;
239             }
240             tableau.ajouter(elt);
241             assertEquals(testName + " tableau[" + nb + "]",
242                     tableau.capacite(), predictedCapacity);
243         }
244     }
245
246     /**
247      * Test method for {@link tableaux.Iterable#ajouter(java.lang.Object)}.
248      */
249     @Test
250     public final void testAjouter()
251     {
252         String testName = new String("Tableau.ajouter(E)");
253         System.out.println(testName);
254         int predictedSize = 0;
255
256         for (String elt : elementsArray)
257         {
258             tableau.ajouter(elt);
259
260             predictedSize++;
261
262             String lastElement = null;
263             for (Iterator<String> itt = tableau.iterator(); itt.hasNext();)
264             {
265                 lastElement = itt.next();
266             }
267
268             assertEquals(testName + " size", predictedSize, tableau.taille());
269             assertEquals(testName + "last elt comparison", elt, lastElement);
270         }
```



```
271     }
272
273     /**
274      * Test method for {@link tableaux.Iterable#retrait(java.lang.Object)}.
275      */
276     @Test
277     public final void testRetrait()
278     {
279         String testName = new String("Tableau.retrait(E)");
280         System.out.println(testName);
281
282         tableau = new Tableau<String>(elementsCollection);
283         int nbElements = elementsArray.length;
284         int nbElementsLeft = nbElements;
285
286         boolean result = compareElements(testName, tableau, elementsCollection);
287         assertTrue(testName + " no more elts to compare", result);
288         // on va retirer des elts de tableau et elementsCollection dans un
289         // ordre alÃ©atoire
290         int[] indexs = shuffledIndexs(nbElements);
291
292         for (int i = 0; i < nbElements; i++)
293         {
294             tableau.retrait(elementsArray[indexs[i]]);
295             elementsCollection.remove(elementsArray[indexs[i]]);
296             nbElementsLeft = elementsCollection.size();
297
298             result = compareElements(testName, tableau, elementsCollection);
299             assertTrue(testName + nbElementsLeft + "elts compared", result);
300         }
301     }
302
303     /**
304      * Test method for {@link tableaux.Iterable#efface()}.
305      */
306     @Test
307     public final void testEfface()
308     {
309         String testName = new String("Tableau.efface()");
310         System.out.println(testName);
311
312         tableau = new Tableau<String>(elementsCollection);
313
314         assertTrue(testName + "tableau initial non vide", tableau.taille() > 0);
315
316         tableau.efface();
317
318         assertEquals(testName + "tableau final vide", tableau.taille(), 0);
319         Iterator<String> it = tableau.iterator();
320         assertFalse(testName + " pas d'elts Ã  itÃ©rer", it.hasNext());
321     }
322
323     /**
324      * Test method for {@link tableaux.Iterable#insertElement(java.lang.Object)}.
325      */
326     @Test
327     public final void testInsertElementE()
328     {
329         String testName = new String("Tableau.insertElement(E)");
330         System.out.println(testName);
331
332         for (String elt : elementsArray)
333         {
334             tableau.insertElement(elt);
335
336             Iterator<String> it = tableau.iterator();
337             assertEquals(testName + " first elt compare", elt, it.next());
338         }
339     }
340
341     /**
342      * Test method for {@link tableaux.Iterable#insertElement(java.lang.Object, int)}.
343      * Ajout Ã  un index invalide dans une collection vide
344      */
345     @Test(expected = IndexOutOfBoundsException.class)
346     public final void testInsertElementEIntInvalidEmpty()
347     {
348         String testName = new String("Tableau.insertElement(E, int)");
349         System.out.println(testName);
350
351         tableau.insertElement("Bonjour", 1);
352
353         fail(testName + " Ajout ds tableau vide Ã  index invalide rÃ©ussi !");
354     }
355
356     /**
357      * Test method for {@link tableaux.Iterable#insertElement(java.lang.Object, int)}.
358      * Ajout Ã  un index invalide dans une collection pleine
359      */
360     @Test(expected = IndexOutOfBoundsException.class)
```

```
361     public final void testInsertElementEIntInvalidFull()
362     {
363         String testName = new String("Tableau.insertElement(E, int)");
364         System.out.println(testName);
365 
366         tableau = new Tableau<String>(elementsCollection);
367 
368         tableau.insertElement("Bonjour", tableau.taille() + 1);
369 
370         fail(testName + " Ajout ds tableau plein Ã  index invalide rÃ©ussi !");
371     }
372 
373     /**
374      * Test method for {@link tableaux.Iterable#insertElement(java.lang.Object, int)}.
375      */
376     @Test
377     public final void testInsertElementEInt()
378     {
379         String testName = new String("Tableau.insertElement(E, int)");
380         System.out.println(testName);
381         int nbElements = elementsArray.length;
382         elementsCollection.clear();
383         int currentSize = 0;
384         boolean result = false;
385 
386         // Ajouts en dÃ©but et fin
387         for (int i = 0; i < (nbElements / 2); i++)
388         {
389             // Ajout au dÃ©but
390             tableau.insertElement(elementsArray[i], 0);
391             elementsCollection.add(0, elementsArray[i]);
392 
393             currentSize = elementsCollection.size();
394 
395             result = compareElements(testName, tableau, elementsCollection);
396             assertTrue(testName + " after push front", result);
397 
398             // Ajout Ã  la fin
399             int sourceIdx = nbElements-1-i;
400             tableau.insertElement(elementsArray[sourceIdx], currentSize);
401             elementsCollection.add(currentSize, elementsArray[sourceIdx]);
402 
403             result = compareElements(testName, tableau, elementsCollection);
404             assertTrue(testName + " after push back", result);
405         }
406 
407         currentSize = elementsCollection.size();
408 
409         // Ajout au milieu
410         String extraElement = "Bonjour";
411         tableau.insertElement(extraElement, currentSize/2);
412         elementsCollection.add(currentSize/2, extraElement);
413 
414         result = compareElements(testName, tableau, elementsCollection);
415         assertTrue(testName + " after push middle", result);
416     }
417 
418     /**
419      * Test method for {@link tableaux.Iterable#iterator()}.
420      */
421     @Test
422     public final void testIterator()
423     {
424         String testName = new String("Tableau.iterator()");
425         System.out.println(testName);
426 
427         // itÃ©rateur sur tableau vide
428         Iterator<String> itt = tableau.iterator();
429         assertFalse(testName + " iterateur sur tableau vide", itt.hasNext());
430 
431         // itÃ©rateur su tableau rempli
432         tableau = new Tableau<String>(elementsCollection);
433         boolean result = compareElements(testName, tableau, elementsCollection);
434         assertTrue(testName, result);
435 
436         // utilisation du remove sans next
437         for (itt = tableau.iterator(); itt.hasNext(); )
438         {
439             try
440             {
441                 itt.remove();
442                 fail(testName + " remove utilisÃ© avec succÃ¨s sans next dans boucle");
443             }
444             catch (IllegalStateException ise)
445             {
446                 // rien, c'est normal
447             }
448             itt.next();
449             itt.remove();
450         }
```



```
451 
452         assertFalse(testName + " iterator terminÃ© fin boucle", itt.hasNext());
453         assertEquals(testName + " tableau vide avec suite remove", 0,
454                 tableau.taille());
455     }
456 
457     /**
458      * Test method for {@link tableaux.Iterable#equals(java.lang.Object)}.
459      */
460     @Test
461     public final void testEqualsObject()
462     {
463         String testName = new String("Tableau.equals(Object)");
464         System.out.println(testName);
465 
466         // Inegalite avec null
467         boolean result = tableau.equals(null);
468         assertFalse(testName + " inequality with null", result);
469 
470         // Egalite avec this
471         assertTrue(testName + " self equality", tableau.equals(tableau));
472 
473         // Egalite avec une copie de soi mÃªme (vide)
474         Tableau<String> other = new Tableau<String>(tableau);
475         assertTrue(testName + " equality with copy", tableau.equals(other));
476 
477         // Inegalite avec tableau de contenu diffÃ©rent
478         for (String elt : elementsArray)
479         {
480             tableau.ajouter(elt);
481         }
482         assertFalse(testName + " content inequality", tableau.equals(other));
483 
484         // Egalite sur contenus identiques
485         for (String elt : elementsArray)
486         {
487             other.ajouter(elt);
488         }
489         assertTrue(testName + " content equality", tableau.equals(other));
490 
491         // Inegalite avec un objet quelconque
492         assertFalse(testName + " type inequality", tableau.equals(new Object()));
493 
494         // Inegalite avec un autre Iterable
495         assertFalse(testName + " inequality with Iterable",
496                 tableau.equals(elementsCollection));
497     }
498 
499     /**
500      * Test method for {@link tableaux.Iterable#hashCode()}.
501      */
502     @Test
503     public final void testHashCode()
504     {
505         String testName = new String("Tableau.hashCode()");
506         System.out.println(testName);
507 
508         // Hash code sur tableau vide
509         assertEquals(testName + " empty tableau", 1, tableau.hashCode());
510 
511         tableau = new Tableau<String>(elementsCollection);
512 
513         // Hash code sur tableau rempli Ã©gal au hascode des collections standard
514         assertEquals(testName + " full tableau", tableau.hashCode(),
515                 elementsCollection.hashCode());
516     }
517 
518     /**
519      * Test method for {@link tableaux.Iterable#toString()}.
520      */
521     @Test
522     public final void testToString()
523     {
524         String testName = new String("Tableau.toString()");
525         System.out.println(testName);
526 
527         tableau = new Tableau<String>(elementsCollection);
528 
529         StringBuilder sb = new StringBuilder();
530         sb.append("[");
531         for (Iterator<String> it = tableau.iterator(); it.hasNext();)
532         {
533             sb.append(it.next().toString());
534             if (it.hasNext())
535             {
536                 sb.append(",");
537             }
538         }
539         sb.append("](");
540         sb.append(Integer.toString(tableau.taille()));
```

```
541             sb.append(",");
542             sb.append(Integer.toString(tableau.capacite()));
543             sb.append(")");
544             String expected = sb.toString();
545 
546             assertEquals(testName, expected, tableau.toString());
547 
548         }
549 
550 }
```



```
1   package tests;
2   
3   import static org.junit.Assert.assertEquals;
4   import static org.junit.Assert.assertFalse;
5   import static org.junit.Assert.assertNotNull;
6   import static org.junit.Assert.assertTrue;
7   import static org.junit.Assert.fail;
8   
9   import java.lang.reflect.InvocationTargetException;
10  import java.util.ArrayList;
11  import java.util.Arrays;
12  import java.util.Collection;
13  import java.util.HashMap;
14  import java.util.Iterator;
15  import java.util.Map;
16  
17  import org.junit.After;
18  import org.junit.AfterClass;
19  import org.junit.Before;
20  import org.junit.BeforeClass;
21  import org.junit.Test;
22  import org.junit.runner.RunWith;
23  import org.junit.runners.Parameterized;
24  import org.junit.runners.Parameterized.Parameters;
25  
26  import ensembles.EnsembleTri;
27  import ensembles.EnsembleTriFactory;
28  import ensembles.EnsembleTriTableau;
29  
30  /**
31   * Classe de test complÃ©mentaire pour tous les types d'ensembles triÃ©s :
32   * {@link ensembles.EnsembleTriVector}, {@link ensembles.EnsembleTriVector2},
33   * {@link ensembles.EnsembleTriListe}, {@link ensembles.EnsembleTriListe2},
34   * {@link ensembles.EnsembleTriTableau}, {@link ensembles.EnsembleTriTableau2}
35   * @author davidroussel
36   */
37  @RunWith(value = Parameterized.class)
38  public class EnsembleTriTest
39  {
40      /**
41       * l'ensemble Ã  tester
42       */
43      private EnsembleTri<String> ensemble;
44  
45      /**
46       * Le type d'ensemble Ã  tester.
47       */
48      private Class<? extends EnsembleTri<String>> typeEnsemble;
49  
50      /**
51       * Nom du type d'ensemble Ã  tester
52       */
53      private String typeName;
54  
55      /**
56       * Les diffÃ©rentes natures d'ensembles Ã  tester
57       */
58      @SuppressWarnings("unchecked")
59      private static final Class<? extends EnsembleTri<String>>[] typesEnsemble =
60      (Class<? extends EnsembleTri<String>>[]) new Class<?>[]
61      {
62          /*
63           * TODO Commenter / dÃ©commenter les lignes ci-dessous en focntion
64           * de votre avancement (Attention la derniÃ¨re ligne non commentÃ©e
65           * ne doit pas avoir de virgule)
66           */
67  //      EnsembleTriVector.class,
68  //      EnsembleTriVector2.class,
69          EnsembleTriTableau.class
70  //      EnsembleTriTableau2.class,
71  //      EnsembleTriListe.class,
72  //      EnsembleTriListe2.class
73      };
74  
75      /**
76       * Elements pour remplir l'ensemble
77       */
78      private static final String[] elements = new String[] {
79              "Lorem",            // 0
80              "ipsum",            // 6
81              "sit",              // 7
82              "dolor",            // 4
83              "amet",             // 2
84              "dolor",            // 4
85              "amet",             // 2
86              "consectetur",      // 3
87              "adipisicing",      // 1
88              "elit"              // 5
89      };
90  
```

```
91      /**
92       * Rang d'insertion des Ã©lÃ©ments successifs
93       */
94      private static final int[] insertionRank = new int[] {
95          0,  // Lorem
96          1,  // ipsum
97          2,  // sit
98          1,  // dolor
99          1,  // amet
100         2,  // dolor
101         1,  // amet
102         2,  // consectetur
103         1,  // adipisicing
104         5,  // elit
105     };
106     /**
107      * Elements triÃ©s pour contrÃ´ler le remplissage de l'ensemble
108      */
109     private static final String[] singleSortedElements = new String[] {
110         "Lorem",            // 0
111         "adipisicing",      // 1
112         "amet",             // 2
113         "consectetur",      // 3
114         "dolor",            // 4
115         "elit",             // 5
116         "ipsum",            // 6
117         "sit"               // 7
118     };
119
120     /**
121      * Elements triÃ©s pour contrÃ´ler le remplissage de l'ensemble
122      */
123     private static final String[][] insertSortedElements = new String[][] {
124         {"Lorem"},
125         {"Lorem", "ipsum"},
126         {"Lorem", "ipsum", "sit"},
127         {"Lorem", "dolor", "ipsum", "sit"},
128         {"Lorem", "amet", "dolor", "ipsum", "sit"},
129         {"Lorem", "amet", "dolor", "ipsum", "sit"},
130         {"Lorem", "amet", "dolor", "ipsum", "sit"},
131         {"Lorem", "amet", "consectetur", "dolor", "ipsum", "sit"},
132         {"Lorem", "adipisicing", "amet", "consectetur", "dolor", "ipsum", "sit"},
133         singleSortedElements
134     };
135
136     /**
137      * Collection pour contenir les Ã©lements de remplissage
138      */
139     private ArrayList<String> listElements;
140
141     /**
142      * Construit une instance de EnsembleTri<String> en fonction d'un type
143      * d'ensemble Ã  crÃ©er et Ã©ventuellement d'un contenu l'ensemble Ã  mettre en
144      * place
145      *
146      * @param testName le message Ã  rÃ©pÃ©ter dans les assertions en fonction du
147      *            test dans lequel est employÃ© cette mÃ©thode
148      * @param type le type d'ensemble Ã  crÃ©er
149      * @param content le contenu Ã  mettre en place dans le nouvel ensemble, ou
150      * bien null si aucun contenu n'est requis.
151      * @return un nouvel ensemble du type demandÃ© evt rempli avec le contenu
152      * fournit s'il est non null.
153      */
154     private static EnsembleTri<String>
155     constructEnsemble(String testName,
156                       Class<? extends EnsembleTri<String>> type,
157                       Iterable<String> content)
158     {
159         EnsembleTri<String> ensemble = null;
160
161         try
162         {
163             ensemble = EnsembleTriFactory.<String>getEnsemble(type, content);
164         }
165         catch (SecurityException e)
166         {
167             fail(testName +  " constructor security exception");
168         }
169         catch (NoSuchMethodException e)
170         {
171             fail(testName +  " constructor not found");
172         }
173         catch (IllegalArgumentException e)
174         {
175             fail(testName + " wrong constructor arguments");
176         }
177         catch (InstantiationException e)
178         {
179             fail(testName + " instanciation exception");
180         }
```



```
181         catch (IllegalAccessException e)
182         {
183             fail(testName + " illegal access");
184         }
185         catch (InvocationTargetException e)
186         {
187             fail(testName + " invocation exception");
188         }
189
190         return ensemble;
191     }
192
193     /**
194      * Compare les Ã©lÃ©ments d'un ensemble pour vÃ©rifier qu'ils sont tous dans
195      * un tableau donnÃ© et dans le mÃªme ordre
196      * @param testName le nom du test dans lequel est utilisÃ© cette mÃ©thode
197      * @param ensemble l'ensemble dont on doit comparer les Ã©lÃ©ments
198      * @param array le tableau utilisÃ© pour vÃ©rifier la prÃ©sence des Ã©lÃ©ments
199      * de l'ensemble
200      * @return true si tous les Ã©lÃ©ments du tableau sont prÃ©sents dans l'ensemble
201      * et dans le mÃªme ordre
202      */
203     private static boolean compareElts2Array(String testName,
204             EnsembleTri<String> ensemble, String[] array)
205     {
206         Iterator<String> ite = ensemble.iterator();
207
208         if (ite ≠ null)
209         {
210             for (int i = 0; (i < array.length) ∧ ite.hasNext(); i++)
211             {
212                 String ensembleElt = ite.next();
213                 String arrayElt = array[i];
214                 boolean check = ensembleElt.equals(arrayElt);
215                 assertTrue(testName + "[" + i + "]= " + arrayElt + " == "
216                         + ensembleElt + " failed", check);
217                 if (¬check)
218                 {
219                     return false;
220                 }
221             }
222             return true;
223         }
224         else
225         {
226             return false;
227         }
228     }
229
230     /**
231      * VÃ©rifie qu'un ensemble ne contient qu'un seul exemplaire de chacun
232      * de ses Ã©lÃ©ments
233      * @param testName le nom du test dans lequel est employÃ©e cette mÃ©thode
234      * @param ensemble l'ensemble Ã  tester
235      * @return true si chaque Ã©lÃ©ment de l'ensemble n'existe qu'Ã  un seul
236      * exemplaire.
237      */
238     private static <E extends Comparable<E>>
239     boolean checkCount(String testName, EnsembleTri<E> ensemble)
240     {
241         Map<E, Integer> wordCount = new HashMap<E, Integer>();
242         for (E elt : ensemble)
243         {
244             if (¬wordCount.containsKey(elt))
245             {
246                 wordCount.put(elt, Integer.valueOf(1));
247             }
248             else
249             {
250                 Integer count = wordCount.get(elt);
251                 count = Integer.valueOf(count.intValue() + 1);
252                 wordCount.put(elt, count);
253             }
254         }
255
256         for (Integer i : wordCount.values())
257         {
258             int countValue = i.intValue();
259             assertEquals(testName + " count check #" + countValue + " failed",
260                     1, countValue);
261             if (countValue ≠ 1)
262             {
263                 return false;
264             }
265         }
266
267         return true;
268     }
269
270     /**
```

```
271      * ParamÃ¨tres Ã  transmettre au constructeur de la classe de test.
272      *
273      * @return une collection de tableaux d'objet contenant les paramÃ¨tres Ã 
274      *         transmettre au constructeur de la classe de test
275      */
276     @Parameters(name = "{index}:{1}")
277     public static Collection<Object[]> data()
278     {
279         Object[][] data = new Object[typesEnsemble.length][2];
280         for (int i = 0; i < typesEnsemble.length; i++)
281         {
282             data[i][0] = typesEnsemble[i];
283             data[i][1] = typesEnsemble[i].getSimpleName();
284         }
285         return Arrays.asList(data);
286     }
287
288     /**
289      * Constructeur paramÃ¤trÃ© par le type d'ensemble Ã  tester.
290      * LancÃ© pour chaque test
291      * @param typeEnsemble le type d'ensemble Ã  qÃ©nÃ©rer
292      * @param le_nom du type d'ensemble Ã  tester (pour le faire apparaÃ®tre
293      * dans le dÃ©roulement des tests).
294      */
295     public EnsembleTriTest(Class<? extends EnsembleTri<String>> typeEnsemble,
296                            String typeEnsembleName)
297     {
298         this.typeEnsemble = typeEnsemble;
299         typeName = typeEnsembleName;
300     }
301
302     /**
303      * Mise en place avant l'ensemble des tests
304      * @throws java.lang.Exception
305      */
306     @BeforeClass
307     public static void setUpBeforeClass() throws Exception
308     {
309         System.out.println("------------------------------------------------");
310         System.out.println("Test des ensembles triÃ©s");
311         System.out.println("------------------------------------------------");
312     }
313
314     /**
315      * Nettoyage aprÃ¨s l'ensemble des tests
316      * @throws java.lang.Exception
317      */
318     @AfterClass
319     public static void tearDownAfterClass() throws Exception
320     {
321         System.out.println("------------------------------------------------");
322         System.out.println("Fin Test des ensembles triÃ©s");
323         System.out.println("------------------------------------------------");
324     }
325
326     /**
327      * Mise en place avant chaque test
328      * @throws java.lang.Exception
329      */
330     @Before
331     public void setUp() throws Exception
332     {
333         ensemble = constructEnsemble("setUp", typeEnsemble, null);
334         assertNotNull("setUp non null instance failed", ensemble);
335
336         listElements = new ArrayList<String>();
337
338         for (String elt : elements)
339         {
340             listElements.add(elt);
341         }
342     }
343
344     /**
345      * Nettoyage aprÃ¨s chaque test
346      * @throws java.lang.Exception
347      */
348     @After
349     public void tearDown() throws Exception
350     {
351         ensemble.efface();
352         ensemble = null;
353         listElements.clear();
354         listElements = null;
355     }
356
357     /**
358      * Test method for
359      * {@link ensembles.EnsembleTriVector#EnsembleTriVector()} or
360      * {@link ensembles.EnsembleTriVector2#EnsembleTriVector2()} or
```



```
361      * {@link ensembles.EnsembleTriListe#EnsembleTriListe()} or
362      * {@link ensembles.EnsembleTriListe2#EnsembleTriListe2()} or
363      * {@link ensembles.EnsembleTriTableau#EnsembleTriTableau()} or
364      * {@link ensembles.EnsembleTriTableau2#EnsembleTriTableau2()}
365      */
366     @Test
367     public final void testDefaultConstructor()
368     {
369         String testName = new String(typeName + "()");
370         System.out.println(testName);
371
372         ensemble = constructEnsemble(testName, typeEnsemble, null);
373         assertNotNull(testName + " non null instance failed" , ensemble);
374
375         assertEquals(testName + " instance type failed", typeEnsemble,
376                 ensemble.getClass());
377         assertTrue(testName +  " empty instance failed", ensemble.estVide());
378         assertEquals(testName + "instance size failed", 0, ensemble.cardinal());
379     }
380
381     /**
382      * Test method for
383      * {@link ensembles.EnsembleTriVector#EnsembleTriVector(Iterable)} or
384      * {@link ensembles.EnsembleTriVector2#EnsembleTriVector2(Iterable)} or
385      * {@link ensembles.EnsembleTriListe#EnsembleTriListe(Iterable)} or
386      * {@link ensembles.EnsembleTriListe2#EnsembleTriListe2(Iterable)} or
387      * {@link ensembles.EnsembleTriTableau#EnsembleTriTableau(Iterable)} or
388      * {@link ensembles.EnsembleTriTableau2#EnsembleTriTableau2(Iterable)} or
389      */
390     @Test
391     public final void testCopyConstructor()
392     {
393         String testName = new String(typeName + "(Iterable)");
394         System.out.println(testName);
395
396         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
397         assertNotNull(testName + " non null instance failed" , ensemble);
398
399         assertEquals(testName + " instance type failed", typeEnsemble,
400                 ensemble.getClass());
401         assertFalse(testName + " not empty instance failed", ensemble.estVide());
402         boolean compare = compareElts2Array(testName, ensemble,
403                 singleSortedElements);
404         assertTrue(testName + " elts compare failed", compare);
405
406         // Tous les Ã©lÃ©ments de ensemble doivent se retrouver dans list
407         for (String elt : ensemble)
408         {
409             assertTrue(testName + "check content [" + elt + "] failed",
410                     listElements.contains(elt));
411         }
412
413         // Tous les Ã©lÃ©ments de l'ensemble n'existent qu'Ã  un seul exemplaire
414         boolean countCheck = EnsembleTriTest.<String>checkCount(testName,
415                 ensemble);
416         assertTrue(testName + "after count check failed", countCheck);
417
418     }
419
420     /**
421      * Test method for {@link ensembles.EnsembleTri#ajout(java.lang.Comparable)}.
422      */
423     @Test
424     public final void testAjout()
425     {
426         String testName = new String(typeName + ".ajout(E)");
427         System.out.println(testName);
428
429         assertTrue(testName + " vide avant remplissage failed",
430                 ensemble.estVide());
431
432         int size = 0;
433         for (int i = 0; i < elements.length; i++)
434         {
435             if (¬ensemble.contient(elements[i]))
436             {
437                 size++;
438             }
439             ensemble.ajout(elements[i]);
440             assertEquals(testName + " size failed", size, ensemble.cardinal());
441             boolean checkElts = compareElts2Array(testName, ensemble,
442                     insertSortedElements[i]);
443             assertTrue(testName + " check elts failed", checkElts);
444         }
445     }
446
447     /**
448      * Test method for {@link ensembles.EnsembleTri#rang(java.lang.Comparable)}.
449      */
450     @Test
```

```
451     public final void testRang()
452     {
453         String testName = new String(typeName + ".rang(E)");
454         System.out.println(testName);
455 
456         assertTrue(testName + " vide avant remplissage failed",
457                 ensemble.estVide());
458 
459         for (int i = 0; i < elements.length; i++)
460         {
461             assertEquals(testName + " rang de " + elements[i] + "[" + i
462                     + "] failed", insertionRank[i], ensemble.rang(elements[i]));
463             ensemble.ajout(elements[i]);
464         }
465     }
466 }
```