```
# 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 -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/EnsembleTest.java \
$(SRC)/tests/EnsembleListeTest.java \
$(SRC)/tests/EnsembleTableauTest.java \
$(SRC)/tests/EnsembleVectorTest.java \
```



```
$(SRC)/tests/EnsembleTriTest.java

OTHER =

.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 $(FIGDIR)/*.pdf

# 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)
     */
    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 ...
     * }
```

```
165       * </code>
166       *
167       * @return un nouvel itÃ©rateur sur la liste
168       * @see {@link Iterable#iterator()}
169       */
170      @Override
171      public abstract Iterator<E> iterator();
172  }
```



```
1  /**
2   * Package contenant la classe {@link tableaux.Tableau} : tableau de donnÃ©es de
3   * taille variable
4   */
5  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
```



```java
     *
     * @param element l'Ã©lÃ©ment Ã  insÃ©rer
     */
    public void ajouter(E element)
    {
        if (size ≥ capacity)
        {
            // ajouterCapacite(Math.max(INCREMENT, (size − capacity) + 1));
            int sc1 = (size − capacity) + 1;
            ajouterCapacite((INCREMENT ≥ sc1 ? INCREMENT : sc1));
        }
        table[size] = element;
        size++;
    }

    /**
     * Ajout de nbCases au tableau
     *
     * @param nbCases nombre de cases Ã  ajouter.
     */
    protected void ajouterCapacite(int nbCases)
    {
        if (nbCases > 0)
        {
            capacity += nbCases;
            @SuppressWarnings("unchecked")
            E[] newTable = (E[]) new Object[capacity];
            for (int i = 0; i < size; i++)
            {
                newTable[i] = table[i];
                table[i] = null; // avoid weak references
            }
            table = newTable;
        }
    }

    /**
     * Retrait de la prmiÃ¨re occurrence d'un Ã©lÃ©ment
     *
     * @param element l'Ã©lÃ©ment Ã  retirer du tableau
     * @return true si l'Ã©lÃ©ment a Ã©tÃ© trouvÃ© et retirÃ©
     */
    public boolean retrait(E element)
    {
        for (Iterator<E> it = iterator(); it.hasNext();)
        {
            if (it.next().equals(element))
            {
                it.remove();
                return true;
            }
        }

        return false;
    }

    /**
     * Effacement de tous les Ã©lÃ©ments tu tableau
     */
    public void efface()
    {
        for (Iterator<E> it = iterator(); it.hasNext();)
        {
            it.next();
            it.remove();
        }
    }

    /**
     * Insertion d'un Ã©lÃ©ment en dÃ©but de tableau
     *
     * @param element l'Ã©lÃ©ment Ã  insÃ©rer
     */
    public void insertElement(E element)
    {
        try
        {
            insertElement(element, 0);
        }
        catch (IndexOutOfBoundsException ioobe)
        {
            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 }
```



```
1  /**
2   * Package contenant la dÃ©finition d'un {@link ensembles.Ensemble} comme Ã©tant
3   * une collection (a priori non ordonnÃ©e, mÃªme si le conteneur sous jacent peut
4   * Ãªtre ordonnÃ©). {@link ensembles.EnsembleGenerique} fournit une implÃ©mentation
5   * partielle des ensembles sans connaÃ®tre encore le conteneur sous-jacent (qui
6   * peut Ãªtre un {@link java.util.Vector}, ou bien une {@link listes.Liste}, ou
7   * encore un {@link tableaux.Tableau}. {@link ensembles.EnsembleGenerique}
8   * n'implÃ©mente pas les opÃ©rations :
9   * <ul>
10  * <li>d'ajout {@link ensembles.EnsembleGenerique#ajout(Object)} puisqu'elle est
11  * spÃ©cifique au conteneur sous-jacent</li>
12  * <li>de construction d'un itÃ©rateur
13  * {@link ensembles.EnsembleGenerique#iterator()} puisqu'elle est aussi
14  * spÃ©cifique au conteneur sous-jacent</li>
15  * <li>les opÃ©rations ensembliste comme
16  * {@link ensembles.Ensemble#union(Ensemble)},
17  * {@link ensembles.Ensemble#intersection(Ensemble)},
18  * {@link ensembles.Ensemble#complement(Ensemble)} et
19  * {@link ensembles.Ensemble#difference(Ensemble)} de part le fait qu'elle est
20  * une classe abstraite et ne peut donc pas "crÃ©er" l'ensemble rÃ©sultat de
21  * l'opÃ©ration ensembliste. En revanche elle propose une implÃ©mentation basÃ©e
22  * sur les mÃ©thodes de classes dans lesquelle l'ensemble rÃ©sultat est dÃ©jÃ  crÃ©Ã©
23  * (par une des classes filles)</li>
24  * </ul>
25  * {@link ensembles.EnsembleGenerique} implÃ©mente donc
26  * <ul>
27  * <li>{@link ensembles.Ensemble#union(Ensemble, Ensemble, Ensemble)}</li>
28  * <li>{@link ensembles.Ensemble#intersection(Ensemble, Ensemble, Ensemble)}</li>
29  * <li>{@link ensembles.Ensemble#complement(Ensemble, Ensemble, Ensemble)}</li>
30  * <li>{@link ensembles.Ensemble#difference(Ensemble, Ensemble, Ensemble)}</li>
31  * </ul>
32  */
33 package ensembles;
```

```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©
```



```java
     *       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)
    {
        /*
         * TODO ComplÃ©ter ...
         */

        return false;
    }

    /**
     * Efface tous les Ã©lements de l'ensemble en utilisant le remove de
     * l'itÃ©rateur fournit par {@link #iterator()}
     */
    public default void efface()
    {
        /*
         * TODO ComplÃ©ter ...
         */
    }

    /**
     * Taille de l'ensemble en utilisant l'itÃ©rateur
     *
     * @return le nombre d'Ã©lÃ©ments dans l'ensemble
     * @see ensembles.Ensemble#cardinal() Attention : si l'on utilise estVide
     *      dans cardinal, il ne faut pas utiliser cardinal dans estVide
     * @note Cette mÃ©thode aura intÃ©rÃªt Ã  Ãªtre rÃ©implÃ©mentÃ©e dans les classes
     *       filles qui utilisent des conteneurs pouvant donner leur taille
     *       directement
     */
    public default int cardinal()
    {
        int count = 0;

        /*
         * TODO ComplÃ©ter ...
         */

        return count;
    }

    /**
     * Union avec un autre ensemble : (this union ensemble).
     *
     * @param ensemble l'autre ensemble avec lequel on veut crÃ©er une union
     * @return un nouvel ensemble contenant l'union de l'ensemble courant et de
     *         l'ensemble passÃ© en argument
     */
    public abstract Ensemble<E> union(Ensemble<E> ensemble);

    /**
     * ImplÃ©mentation de classe de l'union de deux ensemble dans un autre
     * ensemble
     *
     * @param ens1 le premier ensemble
     * @param ens2 le second ensemble
     * @param res l'ensemble contenant l'union de ens1 et ens2
     */
    public static <E> void union(Ensemble<E> ens1, Ensemble<E> ens2, Ensemble<E> res)
    {
        /*
         * TODO ComplÃ©ter ...
         */
    }

    /**
     * Intersection avec un autre ensemble : (this inter ensemble).
     *
     * @param ensemble l'autre ensemble avec lequel on veut crÃ©er une
     *                 intersection
     * @return un nouvel ensemble contenant l'intersection de l'ensemble courant
     *         et de l'ensemble passÃ© en argument
     */
    public abstract Ensemble<E> intersection(Ensemble<E> ensemble);

    /**
```

```
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 }
```

```java
package ensembles;

import java.util.Iterator;

/**
 * Ensemble GÃ©nÃ©rique implÃ©mentant partiellement les opÃ©rations communes Ã  tous
 * les ensembles quels que soit les conteneurs sous-jacents utilisÃ©s pour
 * stocker les Ã©lÃ©ments de l'ensemble. L'ensemble gÃ©nÃ©rique est implÃ©mentÃ© en
 * majeure partie grÃ¢ce Ã  l'itÃ©rateur fourni par la mÃ©thode {@link #iterator()}
 *
 * @author davidroussel
 */
abstract class EnsembleGenerique<E> implements Ensemble<E>
{
    /*
     * (non-Javadoc)
     * @see ensembles.Ensemble#ajout(java.lang.Object)
     */
    @Override
    public abstract boolean ajout(E element);

    /*
     * (non-Javadoc)
     * @see ensembles.Ensemble#union(ensembles.Ensemble)
     */
    @Override
    public abstract Ensemble<E> union(Ensemble<E> ensemble);

    /*
     * (non-Javadoc)
     * @see ensembles.Ensemble#intersection(ensembles.Ensemble)
     */
    @Override
    public abstract Ensemble<E> intersection(Ensemble<E> ensemble);

    /*
     * (non-Javadoc)
     * @see ensembles.Ensemble#complement(ensembles.Ensemble)
     */
    @Override
    public abstract Ensemble<E> complement(Ensemble<E> ensemble);

    /*
     * (non-Javadoc)
     * @see ensembles.Ensemble#iterator()
     */
    @Override
    public abstract Iterator<E> iterator();

    /**
     * Test d'Ã©galitÃ© entre deux ensembles
     *
     * @param o l'objet Ã  comparer
     * @return true si l'objet Ã  comparer est un ensemble et qu'il contient les
     *         mÃªmes Ã©lÃ©ments (pas forcÃ©ment dans le mÃªme ordre). Si les deux
     *         ensembles sont vides on considÃ¨re qu'ils seront Ã©gaux quel que
     *         soit leur type de contenu (dans la mesure oÃ¹ l'on ne peut pas le
     *         dÃ©terminer avec {@link ensembles.Ensemble#typeElements()}
     * @see java.lang.Object#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<?>
         *          - les typeElements() sont identiques ?
         *              - si typeElements des 2 est null :
         *                  ensembles vides ==> true
         *              - sinon - caster obj en (Ensemble<E>)
         *                  - si tous les elts de l'un sont contenus dans l'autre ==> true
         *                  - sinon ==> false
         *          - sinon (types Ã©lÃ©ments diffÃ©rents) ==> false
         *  - sinon obj n'est pas une instance de Ensemble<?> ==> false
         */
        return false;
    }

    /**
```



```java
     * Hashcode d'un ensemble en utilisant l'itÃ©rateur pour parcourir les
     * Ã©lÃ©ments. Le HashCode d'un ensemble doit Ãªtre calculÃ© comme Ã©tant la
     * somme des hashcodes de ses Ã©lÃ©ments afin de ne pas tenir compte de
     * l'ordre des Ã©lÃ©ments dans la collection sous-jacente.
     *
     * @return le hashage d'un ensemble
     * @see java.lang.Object#hashCode()
     */
    @Override
    public int hashCode()
    {
        int result = 0;
        /*
         * TODO ComplÃ©ter ...
         */
        return result;
    }

    /**
     * Affichage des Ã©lement de l'ensemble sous la forme : par exemple pour un
     * ensemble de 3 elts : "[elt1, elt2, elt3]" oÃ¹ eltn reprÃ©sente le toString
     * du niÃ¨me elt.
     *
     * @return une chaine de caractÃ¨re reprÃ©sentant les Ã©lÃ©ments de l'ensemble
     *         sÃ©parÃ© par des virgules et encadrÃ© par des crochets
     * @see java.lang.Object#toString()
     */
    @Override
    public String toString()
    {
        StringBuilder sb = new StringBuilder();
        sb.append("[");
        /*
         * TODO ComplÃ©ter ...
         */
        sb.append("]");

        return new String(sb);
    }
}
```

```java
package ensembles;

import java.util.Iterator;

import tableaux.Tableau;

/**
 * Ensemble Ã  base de tableaux
 *
 * @author davidroussel
 */
public class EnsembleTableau<E> extends EnsembleGenerique<E>
{
    /**
     * Conteneur sous-jacent : un Tableau<E>
     */
    protected Tableau<E> tableau;

    /**
     * Constructeur par dÃ©faut d'un ensemble Ã  base de {@link tableaux.Tableau}
     */
    public EnsembleTableau()
    {
        /*
         * TODO Remplacer par l'initialisation du tableau
         */
        tableau = null;
    }

    /**
     * Constructeur de copie Ã  partir d'un {@link Iterable}
     *
     * @param elements l'itÃ©rable dont on doit copier les Ã©lÃ©ments
     */
    public EnsembleTableau(Iterable<E> elements)
    {
        /*
         * TODO Remplacer par l'initialisation du tableau, puis l'ajout (au
         * sens des ensembles) des Ã©lÃ©ments de "elements"
         */
        tableau = null;
    }

    /**
     * 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
     * Ce qui revient dans le cas prÃ©sent Ã  ajouter un Ã©lÃ©ment au tableau si
     * celui ci n'y 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©)
     * @see ensembles.EnsembleGenerique#ajout(java.lang.Object)
     */
    @Override
    public boolean ajout(E element)
    {
        /*
         * TODO ComplÃ©ter ...
         */
        return false;
    }

    /**
     * Taille de l'ensemble : rÃ©implÃ©mentation en utilisant les propriÃ©tÃ©s du
     * tableau sous-jacent plutÃ´t que l'itÃ©rateur (amÃ©lioration de performances)
     *
     * @return le nombre d'Ã©lÃ©ments dans l'ensemble
     * @see ensembles.EnsembleGenerique#cardinal()
     */
    @Override
    public int cardinal()
    {
        /*
         * TODO Remplacer par une implÃ©mentation plus performante que celle
         * fournie par dÃ©faut par l'interface Ensemble<E>
         */
        return 0;
    }

```



```java
    /**
     * Union avec un autre ensemble en rÃ©utilisant la mÃ©thode de classe union
     * Ã©crite dans l'ensemble GÃ©nÃ©rique (
     * {@link ensembles.EnsembleGenerique#union(ensembles.Ensemble, ensembles.Ensemble, ensembles.Ensemble)}
     * ) et un nouvel {@link ensemble.EnsembleTableau} pour stocker le rÃ©sultat.
     *
     * @param ensemble l'autre ensemble avec lequel on veut crÃ©er une union
     * @return un nouvel ensemble contenant l'union de l'ensemble courant et de
     *         l'ensemble passÃ© en argument
     * @see ensembles.EnsembleGenerique#union(ensembles.Ensemble,
     *      ensembles.Ensemble, ensembles.Ensemble)
     */
    @Override
    public Ensemble<E> union(Ensemble<E> ensemble)
    {
        /*
         * TODO Remplacer par :
         *  - la crÃ©ation d'un nouvel ensemble rÃ©sultat
         *  - l'union de this et ensemble dans rÃ©sultat en rÃ©utilisant ce que
         *    l'on a dÃ©jÃ  Ã©crit
         *  - le renvoi de rÃ©sultat
         */
        return null;
    }

    /**
     * Intersection avec un autre ensemble en rÃ©utilisant la mÃ©thode de classe
     * intersection Ã©crite dans l'ensemble GÃ©nÃ©rique (
     * {@link ensembles.EnsembleGenerique#intersection(ensembles.Ensemble, ensembles.Ensemble, ensembles.Ensemble)}
     * ) et un nouvel {@link ensemble.EnsembleTableau} pour stocker le rÃ©sultat.
     *
     * @param ensemble l'autre ensemble avec lequel on veut crÃ©er une
     *            intersection
     * @return un nouvel ensemble contenant l'intersection de l'ensemble courant
     *         et de l'ensemble passÃ© en argument
     * @see ensembles.EnsembleGenerique#intersection(ensembles.Ensemble,
     *      ensembles.Ensemble, ensembles.Ensemble)
     */
    @Override
    public Ensemble<E> intersection(Ensemble<E> ensemble)
    {
        /*
         * TODO Remplacer par :
         *  - la crÃ©ation d'un nouvel ensemble rÃ©sultat
         *  - l'intersection de this et ensemble dans rÃ©sultat en rÃ©utilisant
         *    ce que l'on a dÃ©jÃ  Ã©crit
         *  - le renvoi de rÃ©sultat
         */
        return null;
    }

    /**
     * ComplÃ©ment avec un autre ensemble en rÃ©utilisant la mÃ©thode de classe
     * complement Ã©crite dans l'ensemble GÃ©nÃ©rique (
     * {@link ensembles.EnsembleGenerique#complement(ensembles.Ensemble, ensembles.Ensemble, ensembles.Ensemble)}
     * ) et un nouvel {@link ensemble.EnsembleTableau} pour stocker le rÃ©sultat.
     *
     * @param ensemble l'autre ensemble avec lequel on veut crÃ©er le complÃ©ment
     * @return un nouvel ensemble contenant uniquement les Ã©lÃ©ments prÃ©sents
     *         dans l'ensemble courant mais PAS dans l'ensemble passÃ© en
     *         argument
     * @see ensembles.EnsembleGenerique#complement(ensembles.Ensemble,
     *      ensembles.Ensemble, ensembles.Ensemble)
     */
    @Override
    public Ensemble<E> complement(Ensemble<E> ensemble)
    {
        /*
         * TODO Remplacer par :
         *  - la crÃ©ation d'un nouvel ensemble rÃ©sultat
         *  - le complÃ©ment de this et ensemble dans rÃ©sultat en rÃ©utilisant
         *    ce que l'on a dÃ©jÃ  Ã©crit
         *  - le renvoi de rÃ©sultat
         */
        return null;
    }

    /**
     * Factory method fournissant un itÃ©rateur sur l'ensemble en rÃ©utilisant
```

```
     * l'itÃ©rateur du tableau sous-jacent
     *
     * @return un nouvel itÃ©rateur sur cet ensemble
     * @see ensembles.EnsembleGenerique#iterator()
     */
    @Override
    public Iterator<E> iterator()
    {
        /*
         * TODO Remplacer par la crÃ©ation d'un iterateur du tableau
         */
        return null;
    }
}
```



```
package ensembles;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

/**
 * Factory permettant de crÃ©er diffÃ©rents types d'ensembles utilisÃ©s dans les
 * tests
 *
 * @author davidroussel
 */
public class EnsembleFactory<E>
{
    /**
     * Obtention d'un nouvel ensemble d'aprÃ¨s le type d'ensemble souhaitÃ© et un
     * contenu (Ã©ventuel) Ã  copier dans le nouvel ensemble
     *
     * @param typeEnsemble le type d'ensemble demandÃ©: soit
     *            {@link ensembles.EnsembleVector}, soit
     *            {@link ensembles.EnsembleListe}, soit
     *            {@link ensembles.EnsembleTableau}
     * @param content le contenu Ã©ventuel Ã  copier dans le nouvel ensemble ( si
     *            celui ci est nul le constructeur par dÃ©faut sera appelÃ©, s'il
     *            est non null, le constructeur de copie sera appelÃ©
     * @return une nouvelle instance de l'ensemble correspondant au type demandÃ©
     * @throws SecurityException Si le SecurityManager ne permet pas l'accÃ¨s au
     *             constructeur demandÃ©
     * @throws NoSuchMethodException Si le constructeur demandÃ© n'existe pas
     * @throws IllegalArgumentException Si le nombre d'arguments fournis au
     *             constructeur n'est pas le bon
     * @throws InstantiationException si la classe demandÃ©e est abstraite
     * @throws IllegalAccessException Si le constructeur demandÃ© est
     *             inaccessible
     * @throws InvocationTargetException si le constructeur invoquÃ© dÃ©clenche
     *             une exception
     */
    @SuppressWarnings("unchecked")
    public static <E> Ensemble<E> getEnsemble(Class<? extends Ensemble<E>> typeEnsemble, Iterable<E>
 content)
            throws SecurityException, NoSuchMethodException, IllegalArgumentException, Instantiation
Exception,
            IllegalAccessException, InvocationTargetException
    {
        Constructor<? extends Ensemble<E>> constructor = null;
        Class<?>[] argumentsTypes = null;
        Object[] arguments = null;
        Object instance = null;

        if (content ≡ null)
        {
            argumentsTypes = new Class<?>[0];
            arguments = new Object[0];
        }
        else
        {
            argumentsTypes = new Class<?>[1];
            argumentsTypes[0] = Iterable.class;
            arguments = new Object[1];
            arguments[0] = content;
        }

        constructor = typeEnsemble.getConstructor(argumentsTypes);

        if (constructor ≠ null)
        {
            instance = constructor.newInstance(arguments);
        }

        return (Ensemble<E>) instance;
    }
}
```

```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;
    }
```



```java
    }

    /**
     * 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 tableaux.Tableau#hashCode() pour un exemple de hashage utilisant
     *      l'ordre des Ã©lÃ©ments
     * @see ensembles.EnsembleGenerique#hashCode()
     */
    @Override
    public int hashCode()
    {
        final int prime = 31;
        int result = 1;
        /*
         * TODO ComplÃ©ter ...
         */
        return result;
    }
}
```

```
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 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  }
```

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



```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;
import ensembles.EnsembleTriTableau;

/**
 * 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
```



```java
    };

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

```
165     /**
166      * Elements non communs Ã  {@value #elements1} et
167      * {@link #elements2}
168      */
169     private static final String[] diffSingleElements = new String[] {
170         "Lorem",
171         "ipsum",
172         "sit",
173         "consectetur",
174         "adipisicing",
175         "elit"
176     };
177 
178     /**
179      * Elements pour remplir l'ensemble avec des doublons pour vÃ©rifier que ceux
180      * ci ne seront pas ajoutÃ©s dans les ensembles
181      */
182     private static final String[] elements = new String[elements1.length
183             + elements2.length];
184 
185     /**
186      * Collection pour contenir les Ã©lements de remplissage
187      */
188     private ArrayList<String> listElements;
189 
190     /**
191      * Construit une instance de Ensemble<String> en fonction d'un type
192      * d'ensemble Ã  crÃ©er et Ã©ventuellement d'un contenu l'ensemble Ã  mettre en
193      * place
194      *
195      * @param testName le message Ã  rÃ©pÃ©ter dans les assertions en fonction du
196      *            test dans lequel est employÃ© cette mÃ©thode
197      * @param type le type d'ensemble Ã  crÃ©er
198      * @param content le contenu Ã  mettre en place dans le nouvel ensemble, ou
199      * bien null si aucun contenu n'est requis.
200      * @return un nouvel ensemble du type demandÃ© evt rempli avec le contenu
201      * fournit s'il est non null.
202      */
203     private static Ensemble<String>
204     constructEnsemble(String testName,
205                       Class<? extends Ensemble<String>> type,
206                       Iterable<String> content)
207     {
208         Ensemble<String> ensemble = null;
209 
210         try
211         {
212             ensemble = EnsembleFactory.<String>getEnsemble(type, content);
213         }
214         catch (SecurityException e)
215         {
216             fail(testName +  " constructor security exception");
217         }
218         catch (NoSuchMethodException e)
219         {
220             fail(testName +  " constructor not found");
221         }
222         catch (IllegalArgumentException e)
223         {
224             fail(testName + " wrong constructor arguments");
225         }
226         catch (InstantiationException e)
227         {
228             fail(testName + " instanciation exception");
229         }
230         catch (IllegalAccessException e)
231         {
232             fail(testName + " illegal access");
233         }
234         catch (InvocationTargetException e)
235         {
236             fail(testName + " invocation exception");
237         }
238 
239         return ensemble;
240     }
241 
242     /**
243      * Compare les Ã©lÃ©ments d'un ensemble pour vÃ©rifier qu'ils sont tous dans
244      * un tableau donnÃ©
245      * @param testName le nom du test dans lequel est utilisÃ© cette mÃ©thode
246      * @param ensemble l'ensemble dont on doit comparer les Ã©lÃ©ments
```



```
247      * @param array le tableau utilisÃ© pour vÃ©rifier la prÃ©sence des Ã©lÃ©ments
248      * de l'ensemble
249      * @return true si tous les Ã©lÃ©ments du tableau sont prÃ©sents dans l'ensemble
250      */
251     private static boolean compareElts2Array(String testName,
252             Ensemble<String> ensemble, String[] array)
253     {
254         for (String elt : array)
255         {
256             boolean contenu = ensemble.contient(elt);
257             assertTrue(testName + " contient(" + elt + ") failed", contenu);
258             if (¬contenu)
259             {
260                 return false;
261             }
262         }
263         return true;
264     }
265 
266     /**
267      * VÃ©rifie qu'un ensemble ne contient qu'un seul exemplaire de chacun
268      * de ses Ã©lÃ©ments
269      * @param testName le nom du test dans lequel est employÃ©e cette mÃ©thode
270      * @param ensemble l'ensemble Ã  tester
271      * @return true si chaque Ã©lÃ©ment de l'ensemble n'existe qu'Ã  un seul
272      * exemplaire.
273      */
274     private static <E> boolean checkCount(String testName, Ensemble<E> ensemble)
275     {
276         Map<E, Integer> wordCount = new HashMap<E, Integer>();
277         for (E elt : ensemble)
278         {
279             if (¬wordCount.containsKey(elt))
280             {
281                 wordCount.put(elt, Integer.valueOf(1));
282             }
283             else
284             {
285                 Integer count = wordCount.get(elt);
286                 count = Integer.valueOf(count.intValue() + 1);
287                 wordCount.put(elt, count);
288             }
289         }
290 
291         for (Integer i : wordCount.values())
292         {
293             int countValue = i.intValue();
294             assertEquals(testName + " count check #" + countValue + " failed",
295                     1, countValue);
296             if (countValue ≠ 1)
297             {
298                 return false;
299             }
300         }
301 
302         return true;
303     }
304 
305     /**
306      * MÃ©lange les Ã©lÃ©ments d'un tableau
307      * @param elements les Ã©lements Ã  mÃ©langer
308      * @return un tableau de mÃªme dimension avec les Ã©lÃ©ments dans un autre
309      * ordre
310      */
311     private static String[] shuffleElements(String[] elements)
312     {
313         List<String> listElements = Arrays.asList(elements);
314 
315         Collections.shuffle(listElements);
316 
317         String[] result = new String[elements.length];
318         int i = 0;
319         for (String elt : listElements)
320         {
321             result[i++] = elt;
322         }
323 
324         return result;
325     }
326 
327     /**
328      * ParamÃ¨tres Ã  transmettre au constructeur de la classe de test.
```

```
     *
     * @return une collection de tableaux d'objet contenant les paramÃ¨tres Ã 
     *         transmettre au constructeur de la classe de test
     */
    @Parameters(name = "{index}:{1}")
    public static Collection<Object[]> data()
    {
        Object[][] data = new Object[typesEnsemble.length][2];
        for (int i = 0; i < typesEnsemble.length; i++)
        {
            data[i][0] = typesEnsemble[i];
            data[i][1] = typesEnsemble[i].getSimpleName();
        }

        return Arrays.asList(data);
    }

    /**
     * Constructeur paramÃªtrÃ© par le type d'ensemble Ã  tester.
     * LancÃ© pour chaque test
     * @param typeEnsemble le type d'ensemble Ã  gÃ©nÃ©rer
     * @param le nom du type d'ensemble Ã  tester (pour le faire apparaÃ®tre
     * dans le dÃ©roulement des tests).
     */
    public AllEnsembleTest(Class<? extends Ensemble<String>> typeEnsemble,
                           String typeEnsembleName)
    {
        this.typeEnsemble = typeEnsemble;
        typeName = typeEnsembleName;
    }

    /**
     * Mise en place avant l'ensemble des tests
     * @throws java.lang.Exception
     */
    @BeforeClass
    public static void setUpBeforeClass() throws Exception
    {
        int j = 0;
        for (int i = 0; i < elements1.length; i++)
        {
            elements[j++] = elements1[i];
        }
        for (int i = 0; i < elements2.length; i++)
        {
            elements[j++] = elements2[i];
        }
        System.out.println("------------------------------------------------");
        System.out.println("Test des ensembles");
        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 des ensembles");
        System.out.println("------------------------------------------------");
    }

    /**
     * Mise en place avant chaque test
     * @throws java.lang.Exception
     */
    @Before
    public void setUp() throws Exception
    {
        ensemble = constructEnsemble("setUp", typeEnsemble, null);
        assertNotNull("setUp non null ensemble failed", ensemble);

        listElements = new ArrayList<String>();
        for (String elt : elements)
        {
            listElements.add(elt);
        }
    }

    /**
```



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

    /**
     * Test method for {@link ensembles.EnsembleVector#EnsembleVector()} or
     * {@link ensembles.EnsembleListe#EnsembleListe()} or
     * {@link ensembles.EnsembleTableau#EnsembleTableau()}
     */
    @Test
    public final void testDefaultConstructor()
    {
        String testName = new String(typeName + "()");
        System.out.println(testName);

        ensemble = constructEnsemble(testName, typeEnsemble, null);
        assertNotNull(testName + " non null instance failed", ensemble);

        assertEquals(testName + " instance type failed", typeEnsemble,
                ensemble.getClass());
        assertTrue(testName +  " empty instance failed", ensemble.estVide());
        assertEquals(testName + " instance size failed", 0, ensemble.cardinal());
    }

    /**
     * Test method for {@link ensembles.EnsembleVector#EnsembleVector(Iterable)}
     * or {@link ensembles.EnsembleListe#EnsembleListe(Iterable)} or
     * {@link ensembles.EnsembleTableau#EnsembleTableau(Iterable)}
     */
    @Test
    public final void testCopyConstructor()
    {
        String testName = new String(typeName + "(Iterable)");
        System.out.println(testName);

        ensemble = constructEnsemble(testName, typeEnsemble, listElements);
        assertNotNull(testName + " non null instance failed", ensemble);

        assertEquals(testName + " instance type failed", typeEnsemble,
                ensemble.getClass());
        assertFalse(testName +  " not empty instance failed", ensemble.estVide());
        boolean compare = compareElts2Array(testName, ensemble, allSingleElements);
        assertTrue(testName + "elts compare failed", compare);

        // Tous les Ã©lÃ©ments de ensemble doivent se retrouver dans list
        for (String elt : ensemble)
        {
            assertTrue(testName + "check content [" + elt + "] failed",
                    listElements.contains(elt));
        }

        // Tous les Ã©lÃ©ments de l'ensemble n'existent qu'Ã  un seul exemplaire
        boolean countCheck = AllEnsembleTest.<String>checkCount(testName, ensemble);

        assertTrue(testName + "after count check failed", countCheck);
    }

    /**
     * Test method for {@link ensembles.Ensemble#ajout(java.lang.Object)}.
     */
    @Test
    public final void testAjout()
    {
        String testName = new String(typeName + ".ajout(E)");
        System.out.println(testName);

        // Ensemble vide avant remplissage
        assertEquals(testName + " ensemble vide failed", 0, ensemble.cardinal());
        int count = 0;
        for (String elt : elements)
        {
            if (¬ensemble.contient(elt))
            {
                count++;
```

```
            }
            ensemble.ajout(elt);
        }
        // Ensemble non vide aprÃ¨s remplissage
        assertEquals(testName + " ensemble rempli failed", count,
                ensemble.cardinal());

        // Verif taille ensemble
        boolean countCheck = AllEnsembleTest.<String>checkCount(testName, ensemble);
        assertTrue(testName + "after count check failed", countCheck);

        // Comparaison des elts avec allSingleElements
        boolean compare = compareElts2Array(testName, ensemble, allSingleElements);
        assertTrue(testName + "elts compare failed", compare);

        // Ajout d'un elt null
        boolean ajoutNull = ensemble.ajout(null);
        assertFalse(testName + " ajout null is true", ajoutNull);
    }

    /**
     * Test method for {@link ensembles.Ensemble#retrait(java.lang.Object)}.
     */
    @Test
    public final void testRetrait()
    {
        String testName = new String(typeName + ".retrait(E)");
        System.out.println(testName);

        ensemble = constructEnsemble(testName, typeEnsemble, listElements);
        assertNotNull(testName + " non null instance failed", ensemble);

        String[] elementsToRemove = shuffleElements(allSingleElements);

        for (String elt : elementsToRemove)
        {
            ensemble.retrait(elt);

            assertFalse(testName + "no more contains " + elt + " failed",
                    ensemble.contient(elt));
        }

        assertTrue(testName + " ensemble vide aprÃ¨s retraits failed",
                ensemble.estVide());
    }

    /**
     * Test method for {@link ensembles.Ensemble#estVide()}.
     */
    @Test
    public final void testEstVide()
    {
        String testName = new String(typeName + ".estVide()");
        System.out.println(testName);

        assertTrue(testName + " ensemble vide failed", ensemble.estVide());
        assertFalse(testName + " ens vide rien Ã  itÃ©rer failed",
                ensemble.iterator().hasNext());

        ensemble = constructEnsemble(testName, typeEnsemble, listElements);
        assertNotNull(testName + " non null instance failed", ensemble);

        assertFalse(testName + " ensemble vide failed", ensemble.estVide());
        assertTrue(testName + " ens non vide iterable failed",
                ensemble.iterator().hasNext());
    }

    /**
     * Test method for {@link ensembles.Ensemble#contient(java.lang.Object)}.
     */
    @Test
    public final void testContientENull()
    {
        String testName = new String(typeName + ".contient((E)null)");
        System.out.println(testName);
        String mot = null;

        // Contient null sur ensemble vide
        assertFalse(testName + " ens vide !contient(null) failed",
                ensemble.contient(mot));

        // remplissage ensemble
```



```
        ensemble = constructEnsemble(testName, typeEnsemble, listElements);
        assertNotNull(testName + " non null instance failed", ensemble);
        assertEquals(testName + " instance remplie failed",
                allSingleElements.length, ensemble.cardinal());

        // Contient null sur ensemble non vide
        assertFalse(testName + " ens plein !contient(null) failed",
                ensemble.contient((String) null));
    }

    /**
     * Test method for {@link ensembles.Ensemble#contient(java.lang.Object)}.
     */
    @Test
    public final void testContientE()
    {
        String testName = new String(typeName + ".contient(E)");
        System.out.println(testName);
        String mot = new String("Bonjour");

        // Contient mot quelconque sur ensemble vide
        assertFalse(testName + " ens vide !contient(" + mot + ") failed",
                ensemble.contient(mot));

        ensemble = constructEnsemble(testName, typeEnsemble, listElements);
        assertNotNull(testName + " non null instance failed", ensemble);

        // Contient mot quelconque sur ensemble non vide
        assertFalse(testName + " ens vide contient(" + mot + ") failed",
                ensemble.contient(mot));

        // Contient mots contenus
        boolean compare = compareElts2Array(testName, ensemble, allSingleElements);
        assertTrue(testName + " elts compare failed", compare);
    }

    /**
     * Test method for {@link ensembles.Ensemble#contient(ensembles.Ensemble)}.
     */
    @Test
    public final void testContientEnsembleNull()
    {
        String testName = new String(typeName + ".contient((Ensemble<E>)null)");
        System.out.println(testName);

        // !Contient ensemble null dans ensemble vide
        assertFalse(testName + "ens vide !contient(null) failed",
                ensemble.contient((Ensemble<String>) null));

        // !Contient ensemble null dans ensemble plein
        ensemble = constructEnsemble(testName, typeEnsemble, listElements);
        assertNotNull(testName + " non null instance failed", ensemble);
        assertEquals(testName + " instance remplie taille failed",
                allSingleElements.length, ensemble.cardinal());

        assertFalse(testName + "ens plein non !contient(null) failed",
                ensemble.contient((Ensemble<String>) null));
    }

    /**
     * Test method for {@link ensembles.Ensemble#contient(ensembles.Ensemble)}.
     */
    @Test
    public final void testContientEnsembleOfE()
    {
        for (int i = 0; i < typesEnsemble.length; i++)
        {
            Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
            String otherTypeName = otherType.getSimpleName();

            String testName = new String(typeName + ".contient("
                    + otherTypeName + "<E>)");
            System.out.println(testName);

            // sous ensemble vide
            Ensemble<String> sousEnsemble = constructEnsemble(testName,
                    typesEnsemble[i], null);
            assertNotNull(testName + " sousEnsemble non null instance failed",
                    sousEnsemble);

            // Contient sous ensemble vide dans ensemble vide
            assertTrue(testName + " ens vide contient sous ens["
```

```
                + typesEnsemble[i].getSimpleName() + "] vide failed",
                ensemble.contient(sousEnsemble));

        // remplissage ensemble
        for (String elt : elements1)
        {
            ensemble.ajout(elt);
        }

        // Contient sous ensemble vide dans ensemble non vide
        assertTrue(testName + " ens plein contient sous ens["
                + typesEnsemble[i].getSimpleName() + "] vide failed",
                ensemble.contient(sousEnsemble));

        // remplissage sous ensemble
        for (int j = 0; j < (elements1.length / 2); j++)
        {
            sousEnsemble.ajout(elements1[j]);
        }

        // Contient sous ensemble non vide ds ens non vide
        assertTrue(testName + " ens plein contient sous ens["
                + typesEnsemble[i].getSimpleName() + "] failed",
                ensemble.contient(sousEnsemble));

        // !Contient sous ensemble non vide non contenu ds ens non vide
        sousEnsemble.ajout("consectetur");
        assertFalse(testName + " ens plein !contient sous ens["
                + typesEnsemble[i].getSimpleName() + "] failed",
                ensemble.contient(sousEnsemble));

        ensemble.efface();
    }
}

/**
 * Test method for {@link ensembles.Ensemble#efface()}.
 */
@Test
public final void testEfface()
{
    String testName = new String(typeName + ".efface()");
    System.out.println(testName);

    assertTrue(testName + "ens vide avant effacement failed",
            ensemble.estVide());

    // Effacement ensemble vide
    ensemble.efface();
    assertTrue(testName + "ens vide aprÃ¨s effacement failed", ensemble.estVide());

    // Effacement ensemble non vide
    ensemble = constructEnsemble(testName, typeEnsemble, listElements);
    assertNotNull(testName + " non null instance failed", ensemble);
    assertFalse(testName + "ens non vide aprÃ¨s remplissage failed",
            ensemble.estVide());
    ensemble.efface();
    assertTrue(testName + "ens vide aprÃ¨s remplissage & effacement failed",
            ensemble.estVide());
}

/**
 * Test method for {@link ensembles.Ensemble#cardinal()}.
 */
@Test
public final void testCardinal()
{
    String testName = new String(typeName + ".cardinal()");
    System.out.println(testName);

    assertTrue(testName + " ensemble vide failed", ensemble.estVide());
    assertEquals(testName + " cardinal 0 sur ensemble vide failed", 0,
            ensemble.cardinal());

    ensemble = constructEnsemble(testName, typeEnsemble, listElements);
    assertNotNull(testName + " non null instance failed", ensemble);

    assertFalse(testName + " ensemble non vide failed", ensemble.estVide());
    assertEquals(testName + " cardinal " + allSingleElements.length
            + " sur ensemble rempli failed", allSingleElements.length,
            ensemble.cardinal());
}
```



```

/**
 * Test method for {@link ensembles.Ensemble#union(ensembles.Ensemble)}.
 */
@Test
public final void testUnion()
{
    for (int i = 0; i < typesEnsemble.length; i++)
    {
        Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
        String otherTypeName = otherType.getSimpleName();

        String testName = new String(typeName + ".union(" + otherTypeName
                + "<E>)");
        System.out.println(testName);

        // remplissage ensemble avec singleElements
        for (String elt : elements1)
        {
            ensemble.ajout(elt);
        }

        // remplissage other avec singleElements2
        Ensemble<String> other = constructEnsemble(testName,
                typesEnsemble[i], null);
        assertNotNull(testName + " other instance non null failed", other);
        for (String elt : elements2)
        {
            other.ajout(elt);
        }

        Ensemble<String> union = ensemble.union(other);

        assertNotNull(testName + " non null union instance failed", union);
        assertFalse(testName + " self union", ensemble ≡ union);
        assertFalse(testName + " self union", other ≡ union);
        assertEquals(testName + " taille failed",
                allSingleElements.length, union.cardinal());
        boolean compare = compareElts2Array(testName, union,
                allSingleElements);
        assertTrue(testName + " elts compare failed", compare);
    }
}

/**
 * Test method for {@link ensembles.Ensemble#intersection(ensembles.Ensemble)}.
 */
@Test
public final void testIntersection()
{
    for (int i = 0; i < typesEnsemble.length; i++)
    {
        Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
        String otherTypeName = otherType.getSimpleName();

        String testName = new String(typeName + ".intersection("
                + otherTypeName + "<E>)");
        System.out.println(testName);

        // remplissage ensemble avec singleElements
        for (String elt : elements1)
        {
            ensemble.ajout(elt);
        }

        // remplissage other avec singleElements2
        Ensemble<String> other = constructEnsemble(testName,
                typesEnsemble[i], null);
        assertNotNull(testName + " other non null instance failed", other);
        for (String elt : elements2)
        {
            other.ajout(elt);
        }

        Ensemble<String> intersection = ensemble.intersection(other);

        assertNotNull(testName + " non null intersection instance failed",
                intersection);
        assertFalse(testName + " self intersection", ensemble ≡ intersection);
        assertFalse(testName + " self intersection", other ≡ intersection);
        assertEquals(testName + " taille failed",
                commonSingleElements.length, intersection.cardinal());
```

```
821             boolean compare = compareElts2Array(testName, intersection,
822                     commonSingleElements);
823             assertTrue(testName + " elts compare failed", compare);
824         }
825     }
826
827     /**
828      * Test method for {@link ensembles.Ensemble#complement(ensembles.Ensemble)}.
829      */
830     @Test
831     public final void testComplement()
832     {
833         for (int i = 0; i < typesEnsemble.length; i++)
834         {
835             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
836             String otherTypeName = otherType.getSimpleName();
837
838             String testName = new String(typeName + ".complement("
839                     + otherTypeName + "<E>)");
840             System.out.println(testName);
841
842             // remplissage ensemble avec singleElements
843             for (String elt : elements1)
844             {
845                 ensemble.ajout(elt);
846             }
847
848             // remplissage other avec singleElements2
849             Ensemble<String> other = constructEnsemble(testName,
850                     typesEnsemble[i], null);
851             assertNotNull(testName + " other non null instance failed", other);
852             for (String elt : elements2)
853             {
854                 other.ajout(elt);
855             }
856
857             Ensemble<String> complement1 = ensemble.complement(other);
858
859             assertNotNull(testName + " non null complement instance 1 failed",
860                     complement1);
861             assertFalse(testName + " self complement1", ensemble ≡ complement1);
862             assertFalse(testName + " self complement1", other ≡ complement1);
863             assertEquals(testName + " taille 1 failed",
864                     complementElements1.length, complement1.cardinal());
865             boolean compare = compareElts2Array(testName, complement1,
866                     complementElements1);
867             assertTrue(testName + " elts compare 1 failed", compare);
868
869             Ensemble<String> complement2 = other.complement(ensemble);
870
871             assertNotNull(testName + " non null complement instance 2 failed",
872                     complement2);
873             assertFalse(testName + " self complement2", ensemble ≡ complement2);
874             assertFalse(testName + " self complement2", other ≡ complement2);
875             assertEquals(testName + " taille 2 failed",
876                     complementElements2.length, complement2.cardinal());
877             compare = compareElts2Array(testName, complement2,
878                     complementElements2);
879             assertTrue(testName + "elts compare 2 failed", compare);
880         }
881     }
882
883     /**
884      * Test method for {@link ensembles.Ensemble#difference(ensembles.Ensemble)}.
885      */
886     @Test
887     public final void testDifference()
888     {
889         for (int i = 0; i < typesEnsemble.length; i++)
890         {
891             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
892             String otherTypeName = otherType.getSimpleName();
893
894             String testName = new String(typeName + ".difference("
895                     + otherTypeName + "<E>)");
896             System.out.println(testName);
897
898             // remplissage ensemble avec singleElements
899             for (String elt : elements1)
900             {
901                 ensemble.ajout(elt);
902             }
```



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

```
            // Equals sur Ensemble mÃªme contenu mÃªme ordre
            assertNotNull(testName + " other non null instance failed", other);
            for(String elt : allSingleElementsSorted)
            {
                other.ajout(elt);
            }
            assertEquals(testName + " ens identique, ordre identique["
                    + otherTypeName + "] failed", ensemble, other);

            // Equals sur Ensemble mÃªme contenu ordre diffÃ©rent
            other.efface();
            for(String elt : allsingleElementsShuffle)
            {
                other.ajout(elt);
            }

            // ensemble est toujours sorted car construit avec
            // allSingleElementsSorted
            if ((ensemble instanceof EnsembleTri<?>) ∧
               ¬(other instanceof EnsembleTri<?>))
            {
                assertFalse(testName + " ens identique, ordre diffÃ©rent["
                        + otherTypeName + "] failed", ensemble.equals(other));
            }
            else
            {
                assertEquals(testName + " ens identique, ordre diffÃ©rent["
                        + otherTypeName + "] failed", ensemble, other);
            }

            // Equals sur Ensemble contenu diffÃ©rent
            other.ajout("bonjour");
            assertFalse(testName + " ens diffÃ©rent failed",
                    ensemble.equals(other));
        }
    }

    /**
     * Test method for {@link ensembles.Ensemble#hashCode()}.
     */
    @Test
    public final void testHashCode()
    {
        String testName = new String(typeName + ".hashCode()");
        System.out.println(testName);
        int hash;
        boolean trie = ensemble instanceof EnsembleTri<?>;
        if (trie)
        {
            hash = 1;
        }
        else
        {
            hash = 0;
        }

        // hash code ensemble vide ==
        //  0 pour les Ensemble
        //  1 pour les EnsembleTri
        assertEquals(testName + " hashcode ens vide failed", hash,
                ensemble.hashCode());

        // hash code ensemble non vide ==
        //  somme des hashcode des elts pour les Ensemble
        //  comme les collections pour les EnsembleTri
        for (String elt : allSingleElements)
        {
            ensemble.ajout(elt);
        }
        if (trie)
        {
            final int prime = 31;
            for (String elt : allSingleElementsSorted)
            {
                hash = (prime * hash) + (elt ≡ null ? 0 : elt.hashCode());
            }
        }
        else
        {
            for (String elt : allSingleElements)
            {
                hash += elt.hashCode();
```



```
            }
        }

        assertEquals(testName + " hashcode ens non vide failed", hash,
                ensemble.hashCode());
    }

    /**
     * Test method for {@link ensembles.Ensemble#toString()}.
     */
    @Test
    public final void testToString()
    {
        String testName = new String(typeName + ".toString()");
        System.out.println(testName);

        ensemble = constructEnsemble(testName, typeEnsemble, listElements);
        assertNotNull(testName + " non null instance failed", ensemble);

        StringBuilder sb = new StringBuilder();
        sb.append("[");
        Iterator<String> it = ensemble.iterator();
        if (it ≠ null)
        {
            for (; it.hasNext();)
            {
                sb.append(it.next().toString());
                if (it.hasNext())
                {
                    sb.append(", ");
                }
            }
            sb.append("]");

            String expected = sb.toString();

            assertEquals(testName, expected, ensemble.toString());
        }
        else
        {
            fail(testName + " null iterator");
        }
    }

    /**
     * Test method for {@link ensembles.Ensemble#iterator()}.
     */
    @Test
    public final void testIterator()
    {
        String testName = new String(typeName + ".iterator()");
        System.out.println(testName);

        Iterator<String> it = null;

        // iterator existe
        it = ensemble.iterator();
        assertNotNull(testName +  " non null instance failed", it);

        // iterator sur ens vide n'a pas d'elts Ã  itÃ©rer
        assertFalse(testName + " !hasNext() sur ens vide failed", it.hasNext());

        // remplissage
        for (String elt : allSingleElements)
        {
            ensemble.ajout(elt);
        }

        it = ensemble.iterator();

        // iterator sur ens rempli
        assertTrue(testName + " hashNext() sur ens rempli failed", it.hasNext());

        String[] array;
        if (ensemble instanceof EnsembleTri<?>)
        {
            array = allSingleElementsSorted;
        }
        else
        {
            array = allSingleElements;
        }
```

```
1149
1150        // comparaison des elts
1151        for(int i = 0; (i < array.length) ∧ it.hasNext(); i++)
1152        {
1153            assertEquals(testName + "check elt:" + array[i] + " failed",
1154                    array[i], it.next());
1155        }
1156
1157        // plus l'elts Ã  itÃ©rer
1158        assertFalse(testName + " !hasNext() fin comparaison failed",
1159                it.hasNext());
1160
1161        // retrait des elts avec l'itÃ©rateur
1162        it = ensemble.iterator();
1163        for (int i = 0; (i < array.length) ∧ it.hasNext(); i++)
1164        {
1165            it.next();
1166            it.remove();
1167            assertFalse(testName + " retrait elt:" + array[i] + " failed",
1168                    ensemble.contient(array[i]));
1169        }
1170
1171        // plus l'elts Ã  itÃ©rer
1172        assertFalse(testName + " !hasNext() fin retrait failed", it.hasNext());
1173        assertTrue(testName + " ens vide aprÃ¨s retraits failed",
1174                ensemble.estVide());
1175    }
1176 }
```



```
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.assertNotSame;
7  import static org.junit.Assert.assertSame;
8  import static org.junit.Assert.assertTrue;
9  import static org.junit.Assert.fail;
10
11 import java.util.ArrayList;
12 import java.util.Iterator;
13 import java.util.NoSuchElementException;
14
15 import org.junit.After;
16 import org.junit.AfterClass;
17 import org.junit.Before;
18 import org.junit.BeforeClass;
19 import org.junit.Test;
20
21 import listes.Liste;
22
23 /**
24  * Classe de test de la liste ChainÃ©e
25  * @author davidroussel
26  */
27 public class ListeTest
   {
29
30      /**
31       * La liste Ã  tester.
32       * La nature du contenu de la liste importe peu du moment qu'il est
33       * homogÃ¨ne : donc n'importe quel type ferait l'affaire.
34       */
35      private Liste<String> liste = null;
36
37      /**
38       * Liste des Ã©lÃ©ments Ã  insÃ©rer dans la liste
39       */
40      private static String[] elements;
41
42      /**
43       * Mise en place avant l'ensemble des tests
44       * @throws java.lang.Exception
45       */
46      @BeforeClass
47      public static void setUpBeforeClass() throws Exception
48      {
49          System.out.println("-------------------------------------------------");
50          System.out.println("Test de la Liste");
51          System.out.println("-------------------------------------------------");
52      }
53
54      /**
55       * Nettoyage aprÃ¨s l'ensemble des tests
56       * @throws java.lang.Exception
57       */
58      @AfterClass
59      public static void tearDownAfterClass() throws Exception
60      {
61          System.out.println("-------------------------------------------------");
62          System.out.println("Fin Test de la Liste");
63          System.out.println("-------------------------------------------------");
64      }
65
66      /**
67       * Mise en place avant chaque test
68       * @throws java.lang.Exception
69       */
70      @Before
71      public void setUp() throws Exception
72      {
73          elements = new String[] {
74                      "Hello",
75                      "Brave",
76                      "New",
77                      "World"
78                  };
79          liste = new Liste<String>();
80      }
81
82      /**
```

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

    /**
     * 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]);
        }
    }

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

        liste.add(elements[0]);

        assertFalse(testName + " ajout 1 elt failed", liste.empty());

        // Ajout null dans une liste non vide (sinon on fait un insere(null))
        // Doit lever une NullPointerException
        liste.add(null);

        fail(testName + " ajout null sans exception");
    }

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

        // Insertion elt null
        try
        {
            liste.insert(null);

            fail(testName +" insertion elt null");
        }
        catch (NullPointerException e)
        {
            assertTrue(testName + " insertion elt null, liste vide failed",
                    liste.empty());
        }

        // Insertion dans une liste vide
        int lastIndex = elements.length - 1;
        liste.insert(elements[lastIndex]);
        assertFalse(testName + "liste non vide failed", liste.empty());
        Iterator<String> it = liste.iterator();
        String insertedElt = it.next();
        assertSame(testName + " contrÃ´le ref element[" + lastIndex + "] failed",
                insertedElt, elements[lastIndex]);
        // 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.insert(elements[lastIndex - i]);

            insertedElt = liste.iterator().next();
            assertSame(testName + " contrÃ´le ref element[" + (lastIndex - i)
                    + "] failed", insertedElt, elements[lastIndex - i]);
        }
    }

    /**
     * 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©atoire
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     /**
```

```
     * Test method for {@link tableaux.Iterable#ajouter(java.lang.Object)}.
     */
    @Test
    public final void testAjouter()
    {
        String testName = new String("Tableau.ajouter(E)");
        System.out.println(testName);
        int predictedSize = 0;

        for (String elt : elementsArray)
        {
            tableau.ajouter(elt);

            predictedSize++;

            String lastElement = null;
            for (Iterator<String> itt = tableau.iterator(); itt.hasNext();)
            {
                lastElement = itt.next();
            }

            assertEquals(testName + " size", predictedSize, tableau.taille());
            assertEquals(testName + "last elt comparison", elt, lastElement);
        }
    }

    /**
     * Test method for {@link tableaux.Iterable#retrait(java.lang.Object)}.
     */
    @Test
    public final void testRetrait()
    {
        String testName = new String("Tableau.retrait(E)");
        System.out.println(testName);

        tableau = new Tableau<String>(elementsCollection);
        int nbElements = elementsArray.length;
        int nbElementsLeft = nbElements;

        boolean result = compareElements(testName, tableau, elementsCollection);
        assertTrue(testName + " no more elts to compare", result);
        // on va retirer des elts de tableau et elementsCollection dans un
        // ordre alÃ©atoire
        int[] indexs = shuffledIndexs(nbElements);

        for (int i = 0; i < nbElements; i++)
        {
            tableau.retrait(elementsArray[indexs[i]]);
            elementsCollection.remove(elementsArray[indexs[i]]);
            nbElementsLeft = elementsCollection.size();

            result = compareElements(testName, tableau, elementsCollection);
            assertTrue(testName + nbElementsLeft + "elts compared", result);
        }
    }

    /**
     * Test method for {@link tableaux.Iterable#efface()}.
     */
    @Test
    public final void testEfface()
    {
        String testName = new String("Tableau.efface()");
        System.out.println(testName);

        tableau = new Tableau<String>(elementsCollection);

        assertTrue(testName + "tableau initial non vide", tableau.taille() > 0);

        tableau.efface();

        assertEquals(testName + "tableau final vide", tableau.taille(), 0);
        Iterator<String> it = tableau.iterator();
        assertFalse(testName + " pas d'elts Ã  itÃ©rer", it.hasNext());
    }

    /**
     * Test method for {@link tableaux.Iterable#insertElement(java.lang.Object)}.
     */
    @Test
    public final void testInsertElementE()
    {
```



```
        String testName = new String("Tableau.insertElement(E)");
        System.out.println(testName);

        for (String elt : elementsArray)
        {
            tableau.insertElement(elt);

            Iterator<String> it = tableau.iterator();
            assertEquals(testName + " first elt compare", elt, it.next());
        }
    }

    /**
     * Test method for {@link tableaux.Iterable#insertElement(java.lang.Object, int)}.
     * Ajout Ã  un index invalide dans une collection vide
     */
    @Test(expected = IndexOutOfBoundsException.class)
    public final void testInsertElementEIntInvalidEmpty()
    {
        String testName = new String("Tableau.insertElement(E, int)");
        System.out.println(testName);

        tableau.insertElement("Bonjour", 1);

        fail(testName + " Ajout ds tableau vide Ã  index invalide rÃ©ussi !");
    }

    /**
     * Test method for {@link tableaux.Iterable#insertElement(java.lang.Object, int)}.
     * Ajout Ã  un index invalide dans une collection pleine
     */
    @Test(expected = IndexOutOfBoundsException.class)
    public final void testInsertElementEIntInvalidFull()
    {
        String testName = new String("Tableau.insertElement(E, int)");
        System.out.println(testName);

        tableau = new Tableau<String>(elementsCollection);

        tableau.insertElement("Bonjour", tableau.taille() + 1);

        fail(testName + " Ajout ds tableau plein Ã  index invalide rÃ©ussi !");
    }

    /**
     * Test method for {@link tableaux.Iterable#insertElement(java.lang.Object, int)}.
     */
    @Test
    public final void testInsertElementEInt()
    {
        String testName = new String("Tableau.insertElement(E, int)");
        System.out.println(testName);
        int nbElements = elementsArray.length;
        elementsCollection.clear();
        int currentSize = 0;
        boolean result = false;

        // Ajouts en dÃ©but et fin
        for (int i = 0; i < (nbElements / 2); i++)
        {
            // Ajout au dÃ©but
            tableau.insertElement(elementsArray[i], 0);
            elementsCollection.add(0, elementsArray[i]);

            currentSize = elementsCollection.size();

            result = compareElements(testName, tableau, elementsCollection);
            assertTrue(testName + " after push front", result);

            // Ajout Ã  la fin
            int sourceIdx = nbElements-1-i;
            tableau.insertElement(elementsArray[sourceIdx], currentSize);
            elementsCollection.add(currentSize, elementsArray[sourceIdx]);

            result = compareElements(testName, tableau, elementsCollection);
            assertTrue(testName + " after push back", result);
        }

        currentSize = elementsCollection.size();

        // Ajout au milieu
        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.Collections;
14  import java.util.HashMap;
15  import java.util.Iterator;
16  import java.util.List;
17  import java.util.Map;
18
19  import org.junit.After;
20  import org.junit.AfterClass;
21  import org.junit.Before;
22  import org.junit.BeforeClass;
23  import org.junit.Test;
24  import org.junit.runner.RunWith;
25  import org.junit.runners.Parameterized;
26  import org.junit.runners.Parameterized.Parameters;
27
28  import ensembles.Ensemble;
29  import ensembles.EnsembleFactory;
30  import ensembles.EnsembleTableau;
31  import ensembles.EnsembleTri;
32
33  /**
34   * Classe de test pour tous les types d'ensembles :
35   * {@link ensembles.EnsembleVector}, {@link ensembles.EnsembleListe},
36   * {@link ensembles.EnsembleTableau}.
37   * Mais aussi pour les mÃ©thodes communes avec les ensemble triÃ©s tels que
38   * {@link ensembles.EnsembleTriVector}, {@link ensembles.EnsembleTriVector2},
39   * {@link ensembles.EnsembleTriListe}, {@link ensembles.EnsembleTriListe2},
40   * {@link ensembles.EnsembleTriTableau}, {@link ensembles.EnsembleTriTableau2}
41   * @author davidroussel
42   */
43  @RunWith(value = Parameterized.class)
44  public class EnsembleTest
    {
46      /**
47       * l'ensemble Ã  tester
48       */
49      private Ensemble<String> ensemble;
50
51      /**
52       * Le type d'ensemble Ã  tester.
53       */
54      private Class<? extends Ensemble<String>> typeEnsemble;
55
56      /**
57       * Nom du type d'ensemble Ã  tester
58       */
59      private String typeName;
60
61      /**
62       * Les diffÃ©rentes natures d'ensembles Ã  tester
63       */
64      @SuppressWarnings("unchecked")
65      private static final Class<? extends Ensemble<String>>[] typesEnsemble =
66      (Class<? extends Ensemble<String>>[]) new Class<?>[]
67      {
68          /*
69           * TODO Commenter / dÃ©commenter les lignes ci-dessous en fonction
70           * de votre avancement (Attention la derniÃ¨re ligne non commentÃ©e
71           * ne doit pas avoir de virgule)
72           */
73          EnsembleTableau.class
74  //      EnsembleVector.class,
75  //      EnsembleListe.class
76      };
77
78      /**
79       * Elements pour remplir l'ensemble : "Lorem ipsum dolor sit amet"
80       */
81      private static final String[] elements1 = new String[] {
82          "Lorem",
```



```
83          "ipsum",
84          "sit",
85          "dolor",
86          "amet"
87      };
88
89      /**
90       * Autres Elements pour remplir un ensemble :
91       * "dolor amet consectetur adipisicing elit"
92       */
93      private static final String[] elements2 = new String[] {
94          "dolor",
95          "amet",
96          "consectetur",
97          "adipisicing",
98          "elit"
99      };
100
101     /**
102      * Elements union de {@value #elements1} et {@link #elements2}
103      */
104     private static final String[] allSingleElements = new String[] {
105         "Lorem",
106         "ipsum",
107         "sit",
108         "dolor",
109         "amet",
110         "consectetur",
111         "adipisicing",
112         "elit"
113     };
114
115     /**
116      * Elements union triÃ©e de {@value #elements1} et
117      * {@link #elements2}
118      */
119     private static final String[] allSingleElementsSorted = new String[] {
120         "Lorem",
121         "adipisicing",
122         "amet",
123         "consectetur",
124         "dolor",
125         "elit",
126         "ipsum",
127         "sit"
128     };
129
130     /**
131      * Elements communs Ã  {@value #elements1} et {@link #elements2}
132      */
133     private static final String[] commonSingleElements = new String[] {
134         "dolor",
135         "amet"
136     };
137
138     /**
139      * Elements du complement de {@value #elements1} et
140      * {@link #elements2}
141      */
142     private static final String[] complementElements1 = new String[] {
143         "Lorem",
144         "ipsum",
145         "sit"
146     };
147
148     /**
149      * Elements du complement de {@value #elements2} et
150      * {@link #elements1}
151      */
152     private static final String[] complementElements2 = new String[] {
153         "consectetur",
154         "adipisicing",
155         "elit"
156     };
157
158     /**
159      * Elements non communs Ã  {@value #elements1} et
160      * {@link #elements2}
161      */
162     private static final String[] diffSingleElements = new String[] {
163         "Lorem",
164         "ipsum",
```

```
165             "sit",
166             "consectetur",
167             "adipisicing",
168             "elit"
169         };
170
171         /**
172          * Elements pour remplir l'ensemble avec des doublons pour vÃ©rifier que ceux
173          * ci ne seront pas ajoutÃ©s dans les ensembles
174          */
175         private static final String[] elements = new String[elements1.length
176                 + elements2.length];
177
178         /**
179          * Collection pour contenir les Ã©lements de remplissage
180          */
181         private ArrayList<String> listElements;
182
183         /**
184          * Construit une instance de Ensemble<String> en fonction d'un type
185          * d'ensemble Ã  crÃ©er et Ã©ventuellement d'un contenu l'ensemble Ã  mettre en
186          * place
187          *
188          * @param testName le message Ã  rÃ©pÃ©ter dans les assertions en fonction du
189          *                 test dans lequel est employÃ© cette mÃ©thode
190          * @param type le type d'ensemble Ã  crÃ©er
191          * @param content le contenu Ã  mettre en place dans le nouvel ensemble, ou
192          * bien null si aucun contenu n'est requis.
193          * @return un nouvel ensemble du type demandÃ© evt rempli avec le contenu
194          * fournit s'il est non null.
195          */
196         private static Ensemble<String>
197         constructEnsemble(String testName,
198                           Class<? extends Ensemble<String>> type,
199                           Iterable<String> content)
200         {
201             Ensemble<String> ensemble = null;
202
203             try
204             {
205                 ensemble = EnsembleFactory.<String>getEnsemble(type, content);
206             }
207             catch (SecurityException e)
208             {
209                 fail(testName +  " constructor security exception");
210             }
211             catch (NoSuchMethodException e)
212             {
213                 fail(testName +  " constructor not found");
214             }
215             catch (IllegalArgumentException e)
216             {
217                 fail(testName + " wrong constructor arguments");
218             }
219             catch (InstantiationException e)
220             {
221                 fail(testName + " instanciation exception");
222             }
223             catch (IllegalAccessException e)
224             {
225                 fail(testName + " illegal access");
226             }
227             catch (InvocationTargetException e)
228             {
229                 fail(testName + " invocation exception");
230             }
231
232             return ensemble;
233         }
234
235         /**
236          * Compare les Ã©lÃ©ments d'un ensemble pour vÃ©rifier qu'ils sont tous dans
237          * un tableau donnÃ©
238          * @param testName le nom du test dans lequel est utilisÃ© cette mÃ©thode
239          * @param ensemble l'ensemble dont on doit comparer les Ã©lÃ©ments
240          * @param array le tableau utilisÃ© pour vÃ©rifier la prÃ©sence des Ã©lÃ©ments
241          * de l'ensemble
242          * @return true si tous les Ã©lÃ©ments du tableau sont prÃ©sents dans l'ensemble
243          */
244         private static boolean compareElts2Array(String testName,
245                 Ensemble<String> ensemble, String[] array)
246         {
```



```
247             for (String elt : array)
248             {
249                 boolean contenu = ensemble.contient(elt);
250                 assertTrue(testName + " contient(" + elt + ") failed", contenu);
251                 if (¬contenu)
252                 {
253                     return false;
254                 }
255             }
256             return true;
257         }
258
259         /**
260          * VÃ©rifie qu'un ensemble ne contient qu'un seul exemplaire de chacun
261          * de ses Ã©lÃ©ments
262          * @param testName le nom du test dans lequel est employÃ©e cette mÃ©thode
263          * @param ensemble l'ensemble Ã  tester
264          * @return true si chaque Ã©lÃ©ment de l'ensemble n'existe qu'Ã  un seul
265          * exemplaire.
266          */
267         private static <E> boolean checkCount(String testName, Ensemble<E> ensemble)
268         {
269             Map<E, Integer> wordCount = new HashMap<E, Integer>();
270             for (E elt : ensemble)
271             {
272                 if (¬wordCount.containsKey(elt))
273                 {
274                     wordCount.put(elt, Integer.valueOf(1));
275                 }
276                 else
277                 {
278                     Integer count = wordCount.get(elt);
279                     count = Integer.valueOf(count.intValue() + 1);
280                     wordCount.put(elt, count);
281                 }
282             }
283
284             for (Integer i : wordCount.values())
285             {
286                 int countValue = i.intValue();
287                 assertEquals(testName + " count check #" + countValue + " failed",
288                         1, countValue);
289                 if (countValue ≠ 1)
290                 {
291                     return false;
292                 }
293             }
294
295             return true;
296         }
297
298         /**
299          * MÃ©lange les Ã©lÃ©ments d'un tableau
300          * @param elements les Ã©lements Ã  mÃ©langer
301          * @return un tableau de mÃªme dimension avec les Ã©lÃ©ments dans un autre
302          * ordre
303          */
304         private static String[] shuffleElements(String[] elements)
305         {
306             List<String> listElements = Arrays.asList(elements);
307
308             Collections.shuffle(listElements);
309
310             String[] result = new String[elements.length];
311             int i = 0;
312             for (String elt : listElements)
313             {
314                 result[i++] = elt;
315             }
316
317             return result;
318         }
319
320         /**
321          * ParamÃ¨tres Ã  transmettre au constructeur de la classe de test.
322          *
323          * @return une collection de tableaux d'objet contenant les paramÃ¨tres Ã 
324          *         transmettre au constructeur de la classe de test
325          */
326         @Parameters(name = "{index}:{1}")
327         public static Collection<Object[]> data()
328         {
```

```
329         Object[][] data = new Object[typesEnsemble.length][2];
330         for (int i = 0; i < typesEnsemble.length; i++)
331         {
332             data[i][0] = typesEnsemble[i];
333             data[i][1] = typesEnsemble[i].getSimpleName();
334         }
335         return Arrays.asList(data);
336     }
337 
338     /**
339      * Constructeur paramÃªtrÃ© par le type d'ensemble Ã  tester.
340      * LancÃ© pour chaque test
341      * @param typeEnsemble le type d'ensemble Ã  gÃ©nÃ©rer
342      * @param le nom du type d'ensemble Ã  tester (pour le faire apparaÃ®tre
343      * dans le dÃ©roulement des tests).
344      */
345     public EnsembleTest(Class<? extends Ensemble<String>> typeEnsemble,
346                         String typeEnsembleName)
347     {
348         this.typeEnsemble = typeEnsemble;
349         typeName = typeEnsembleName;
350     }
351 
352     /**
353      * Mise en place avant l'ensemble des tests
354      * @throws java.lang.Exception
355      */
356     @BeforeClass
357     public static void setUpBeforeClass() throws Exception
358     {
359         int j = 0;
360         for (int i = 0; i < elements1.length; i++)
361         {
362             elements[j++] = elements1[i];
363         }
364         for (int i = 0; i < elements2.length; i++)
365         {
366             elements[j++] = elements2[i];
367         }
368         System.out.println("------------------------------------------------------");
369         System.out.println("Test des ensembles");
370         System.out.println("------------------------------------------------------");
371     }
372 
373     /**
374      * Nettoyage aprÃ¨s l'ensemble des tests
375      * @throws java.lang.Exception
376      */
377     @AfterClass
378     public static void tearDownAfterClass() throws Exception
379     {
380         System.out.println("------------------------------------------------------");
381         System.out.println("Fin Test des ensembles");
382         System.out.println("------------------------------------------------------");
383     }
384 
385     /**
386      * Mise en place avant chaque test
387      * @throws java.lang.Exception
388      */
389     @Before
390     public void setUp() throws Exception
391     {
392         ensemble = constructEnsemble("setUp", typeEnsemble, null);
393         assertNotNull("setUp non null ensemble failed", ensemble);
394 
395         listElements = new ArrayList<String>();
396         for (String elt : elements)
397         {
398             listElements.add(elt);
399         }
400     }
401 
402     /**
403      * Nettoyage aprÃ¨s chaque test
404      * @throws java.lang.Exception
405      */
406     @After
407     public void tearDown() throws Exception
408     {
409         ensemble.efface();
410         ensemble = null;
```



```
411         listElements.clear();
412         listElements = null;
413     }
414 
415     /**
416      * Test method for {@link ensembles.EnsembleVector#EnsembleVector()} or
417      * {@link ensembles.EnsembleListe#EnsembleListe()} or
418      * {@link ensembles.EnsembleTableau#EnsembleTableau()}
419      */
420     @Test
421     public final void testDefaultConstructor()
422     {
423         String testName = new String(typeName + "()");
424         System.out.println(testName);
425 
426         ensemble = constructEnsemble(testName, typeEnsemble, null);
427         assertNotNull(testName + " non null instance failed", ensemble);
428 
429         assertEquals(testName + " instance type failed", typeEnsemble,
430                 ensemble.getClass());
431         assertTrue(testName +  " empty instance failed", ensemble.estVide());
432         assertEquals(testName + " instance size failed", 0, ensemble.cardinal());
433     }
434 
435     /**
436      * Test method for {@link ensembles.EnsembleVector#EnsembleVector(Iterable)}
437      * or {@link ensembles.EnsembleListe#EnsembleListe(Iterable)} or
438      * {@link ensembles.EnsembleTableau#EnsembleTableau(Iterable)}
439      */
440     @Test
441     public final void testCopyConstructor()
442     {
443         String testName = new String(typeName + "(Iterable)");
444         System.out.println(testName);
445 
446         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
447         assertNotNull(testName + " non null instance failed", ensemble);
448 
449         assertEquals(testName + " instance type failed", typeEnsemble,
450                 ensemble.getClass());
451         assertFalse(testName +  " not empty instance failed", ensemble.estVide());
452         boolean compare = compareElts2Array(testName, ensemble, allSingleElements);
453         assertTrue(testName + "elts compare failed", compare);
454 
455         // Tous les Ã©lÃ©ments de ensemble doivent se retrouver dans list
456         for (String elt : ensemble)
457         {
458             assertTrue(testName + "check content [" + elt + "] failed",
459                     listElements.contains(elt));
460         }
461 
462         // Tous les Ã©lÃ©ments de l'ensemble n'existent qu'Ã  un seul exemplaire
463         boolean countCheck = EnsembleTest.<String>checkCount(testName, ensemble);
464 
465         assertTrue(testName + "after count check failed", countCheck);
466     }
467 
468     /**
469      * Test method for {@link ensembles.Ensemble#ajout(java.lang.Object)}.
470      */
471     @Test
472     public final void testAjout()
473     {
474         String testName = new String(typeName + ".ajout(E)");
475         System.out.println(testName);
476 
477         // Ensemble vide avant remplissage
478         assertEquals(testName + " ensemble vide failed", 0, ensemble.cardinal());
479         int count = 0;
480         for (String elt : elements)
481         {
482             if (¬ensemble.contient(elt))
483             {
484                 count++;
485             }
486             ensemble.ajout(elt);
487         }
488         // Ensemble non vide aprÃ¨s remplissage
489         assertEquals(testName + " ensemble rempli failed", count,
490                 ensemble.cardinal());
491 
492         // Verif taille ensemble
```

```
493         boolean countCheck = EnsembleTest.<String>checkCount(testName, ensemble);
494         assertTrue(testName + "after count check failed", countCheck);
495 
496         // Comparaison des elts avec allSingleElements
497         boolean compare = compareElts2Array(testName, ensemble, allSingleElements);
498         assertTrue(testName + "elts compare failed", compare);
499 
500         // Ajout d'un elt null
501         boolean ajoutNull = ensemble.ajout(null);
502         assertFalse(testName + " ajout null is true", ajoutNull);
503     }
504 
505     /**
506      * Test method for {@link ensembles.Ensemble#retrait(java.lang.Object)}.
507      */
508     @Test
509     public final void testRetrait()
510     {
511         String testName = new String(typeName + ".retrait(E)");
512         System.out.println(testName);
513 
514         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
515         assertNotNull(testName + " non null instance failed", ensemble);
516 
517         String[] elementsToRemove = shuffleElements(allSingleElements);
518 
519         for (String elt : elementsToRemove)
520         {
521             ensemble.retrait(elt);
522 
523             assertFalse(testName + "no more contains " + elt + " failed",
524                     ensemble.contient(elt));
525         }
526 
527         assertTrue(testName + " ensemble vide aprÃ¨s retraits failed",
528                 ensemble.estVide());
529     }
530 
531     /**
532      * Test method for {@link ensembles.Ensemble#estVide()}.
533      */
534     @Test
535     public final void testEstVide()
536     {
537         String testName = new String(typeName + ".estVide()");
538         System.out.println(testName);
539 
540         assertTrue(testName + " ensemble vide failed", ensemble.estVide());
541         assertFalse(testName + " ens vide rien Ã  itÃ©rer failed",
542                 ensemble.iterator().hasNext());
543 
544         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
545         assertNotNull(testName + " non null instance failed", ensemble);
546 
547         assertFalse(testName + " ensemble vide failed", ensemble.estVide());
548         assertTrue(testName + " ens non vide iterable failed",
549                 ensemble.iterator().hasNext());
550     }
551 
552     /**
553      * Test method for {@link ensembles.Ensemble#contient(java.lang.Object)}.
554      */
555     @Test
556     public final void testContientENull()
557     {
558         String testName = new String(typeName + ".contient((E)null)");
559         System.out.println(testName);
560         String mot = null;
561 
562         // Contient null sur ensemble vide
563         assertFalse(testName + " ens vide !contient(null) failed",
564                 ensemble.contient(mot));
565 
566         // remplissage ensemble
567         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
568         assertNotNull(testName + " non null instance failed", ensemble);
569         assertEquals(testName + " instance remplie failed",
570                 allSingleElements.length, ensemble.cardinal());
571 
572         // Contient null sur ensemble non vide
573         assertFalse(testName + " ens plein !contient(null) failed",
574                 ensemble.contient((String) null));
```



```
575     }
576 
577     /**
578      * Test method for {@link ensembles.Ensemble#contient(java.lang.Object)}.
579      */
580     @Test
581     public final void testContientE()
582     {
583         String testName = new String(typeName + ".contient(E)");
584         System.out.println(testName);
585         String mot = new String("Bonjour");
586 
587         // Contient mot quelconque sur ensemble vide
588         assertFalse(testName + " ens vide !contient(" + mot + ") failed",
589                 ensemble.contient(mot));
590 
591         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
592         assertNotNull(testName + " non null instance failed", ensemble);
593 
594         // Contient mot quelconque sur ensemble non vide
595         assertFalse(testName + " ens vide contient(" + mot + ") failed",
596                 ensemble.contient(mot));
597 
598         // Contient mots contenus
599         boolean compare = compareElts2Array(testName, ensemble, allSingleElements);
600         assertTrue(testName + " elts compare failed", compare);
601     }
602 
603     /**
604      * Test method for {@link ensembles.Ensemble#contient(ensembles.Ensemble)}.
605      */
606     @Test
607     public final void testContientEnsembleNull()
608     {
609         String testName = new String(typeName + ".contient((Ensemble<E>)null)");
610         System.out.println(testName);
611 
612         // !Contient ensemble null dans ensemble vide
613         assertFalse(testName + "ens vide !contient(null) failed",
614                 ensemble.contient((Ensemble<String>) null));
615 
616         // !Contient ensemble null dans ensemble plein
617         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
618         assertNotNull(testName + " non null instance failed", ensemble);
619         assertEquals(testName + " instance remplie taille failed",
620                 allSingleElements.length, ensemble.cardinal());
621 
622         assertFalse(testName + "ens plein non !contient(null) failed",
623                 ensemble.contient((Ensemble<String>) null));
624     }
625 
626     /**
627      * Test method for {@link ensembles.Ensemble#contient(ensembles.Ensemble)}.
628      */
629     @Test
630     public final void testContientEnsembleOfE()
631     {
632         for (int i = 0; i < typesEnsemble.length; i++)
633         {
634             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
635             String otherTypeName = otherType.getSimpleName();
636 
637             String testName = new String(typeName + ".contient("
638                     + otherTypeName + "<E>)");
639             System.out.println(testName);
640 
641             // sous ensemble vide
642             Ensemble<String> sousEnsemble = constructEnsemble(testName,
643                     typesEnsemble[i], null);
644             assertNotNull(testName + " sousEnsemble non null instance failed",
645                     sousEnsemble);
646 
647             // Contient sous ensemble vide dans ensemble vide
648             assertTrue(testName + " ens vide contient sous ens["
649                     + typesEnsemble[i].getSimpleName() + "] vide failed",
650                     ensemble.contient(sousEnsemble));
651 
652             // remplissage ensemble
653             for (String elt : elements1)
654             {
655                 ensemble.ajout(elt);
656             }
```

```
657
658             // Contient sous ensemble vide dans ensemble non vide
659             assertTrue(testName + " ens plein contient sous ens["
660                     + typesEnsemble[i].getSimpleName() + "] vide failed",
661                     ensemble.contient(sousEnsemble));
662
663             // remplissage sous ensemble
664             for (int j = 0; j < (elements1.length / 2); j++)
665             {
666                 sousEnsemble.ajout(elements1[j]);
667             }
668
669             // Contient sous ensemble non vide ds ens non vide
670             assertTrue(testName + " ens plein contient sous ens["
671                     + typesEnsemble[i].getSimpleName() + "] failed",
672                     ensemble.contient(sousEnsemble));
673
674             // !Contient sous ensemble non vide non contenu ds ens non vide
675             sousEnsemble.ajout("consectetur");
676             assertFalse(testName + " ens plein !contient sous ens["
677                     + typesEnsemble[i].getSimpleName() + "] failed",
678                     ensemble.contient(sousEnsemble));
679
680             ensemble.efface();
681         }
682     }
683
684     /**
685      * Test method for {@link ensembles.Ensemble#efface()}.
686      */
687     @Test
688     public final void testEfface()
689     {
690         String testName = new String(typeName + ".efface()");
691         System.out.println(testName);
692
693         assertTrue(testName + "ens vide avant effacement failed",
694                 ensemble.estVide());
695
696         // Effacement ensemble vide
697         ensemble.efface();
698         assertTrue(testName + "ens vide aprÃ¨s effacement failed", ensemble.estVide());
699
700         // Effacement ensemble non vide
701         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
702         assertNotNull(testName + " non null instance failed", ensemble);
703         assertFalse(testName + "ens non vide aprÃ¨s remplissage failed",
704                 ensemble.estVide());
705         ensemble.efface();
706         assertTrue(testName + "ens vide aprÃ¨s remplissage & effacement failed",
707                 ensemble.estVide());
708     }
709
710     /**
711      * Test method for {@link ensembles.Ensemble#cardinal()}.
712      */
713     @Test
714     public final void testCardinal()
715     {
716         String testName = new String(typeName + ".cardinal()");
717         System.out.println(testName);
718
719         assertTrue(testName + " ensemble vide failed", ensemble.estVide());
720         assertEquals(testName + " cardinal 0 sur ensemble vide failed", 0,
721                 ensemble.cardinal());
722
723         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
724         assertNotNull(testName + " non null instance failed", ensemble);
725
726         assertFalse(testName + " ensemble non vide failed", ensemble.estVide());
727         assertEquals(testName + " cardinal " + allSingleElements.length
728                 + " sur ensemble rempli failed", allSingleElements.length,
729                 ensemble.cardinal());
730     }
731
732     /**
733      * Test method for {@link ensembles.Ensemble#union(ensembles.Ensemble)}.
734      */
735     @Test
736     public final void testUnion()
737     {
738         for (int i = 0; i < typesEnsemble.length; i++)
```



```
739         {
740             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
741             String otherTypeName = otherType.getSimpleName();
742
743             String testName = new String(typeName + ".union(" + otherTypeName
744                     + "<E>)");
745             System.out.println(testName);
746
747             // remplissage ensemble avec singleElements
748             for (String elt : elements1)
749             {
750                 ensemble.ajout(elt);
751             }
752
753             // remplissage other avec singleElements2
754             Ensemble<String> other = constructEnsemble(testName,
755                     typesEnsemble[i], null);
756             assertNotNull(testName + " other instance non null failed", other);
757             for (String elt : elements2)
758             {
759                 other.ajout(elt);
760             }
761
762             Ensemble<String> union = ensemble.union(other);
763
764             assertNotNull(testName + " non null union instance failed", union);
765             assertFalse(testName + " self union", ensemble == union);
766             assertFalse(testName + " self union", other == union);
767             assertEquals(testName + " taille failed",
768                     allSingleElements.length, union.cardinal());
769             boolean compare = compareElts2Array(testName, union,
770                     allSingleElements);
771             assertTrue(testName + " elts compare failed", compare);
772         }
773     }
774
775     /**
776      * Test method for {@link ensembles.Ensemble#intersection(ensembles.Ensemble)}.
777      */
778     @Test
779     public final void testIntersection()
780     {
781         for (int i = 0; i < typesEnsemble.length; i++)
782         {
783             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
784             String otherTypeName = otherType.getSimpleName();
785
786             String testName = new String(typeName + ".intersection("
787                     + otherTypeName + "<E>)");
788             System.out.println(testName);
789
790             // remplissage ensemble avec singleElements
791             for (String elt : elements1)
792             {
793                 ensemble.ajout(elt);
794             }
795
796             // remplissage other avec singleElements2
797             Ensemble<String> other = constructEnsemble(testName,
798                     typesEnsemble[i], null);
799             assertNotNull(testName + " other non null instance failed", other);
800             for (String elt : elements2)
801             {
802                 other.ajout(elt);
803             }
804
805             Ensemble<String> intersection = ensemble.intersection(other);
806
807             assertNotNull(testName + " non null intersection instance failed",
808                     intersection);
809             assertFalse(testName + " self intersection", ensemble == intersection);
810             assertFalse(testName + " self intersection", other == intersection);
811             assertEquals(testName + " taille failed",
812                     commonSingleElements.length, intersection.cardinal());
813             boolean compare = compareElts2Array(testName, intersection,
814                     commonSingleElements);
815             assertTrue(testName + " elts compare failed", compare);
816         }
817     }
818
819     /**
820      * Test method for {@link ensembles.Ensemble#complement(ensembles.Ensemble)}.
```

```
821      */
822     @Test
823     public final void testComplement()
824     {
825         for (int i = 0; i < typesEnsemble.length; i++)
826         {
827             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
828             String otherTypeName = otherType.getSimpleName();
829 
830             String testName = new String(typeName + ".complement("
831                     + otherTypeName + "<E>)");
832             System.out.println(testName);
833 
834             // remplissage ensemble avec singleElements
835             for (String elt : elements1)
836             {
837                 ensemble.ajout(elt);
838             }
839 
840             // remplissage other avec singleElements2
841             Ensemble<String> other = constructEnsemble(testName,
842                     typesEnsemble[i], null);
843             assertNotNull(testName + " other non null instance failed", other);
844             for (String elt : elements2)
845             {
846                 other.ajout(elt);
847             }
848 
849             Ensemble<String> complement1 = ensemble.complement(other);
850 
851             assertNotNull(testName + " non null complement instance 1 failed",
852                     complement1);
853             assertFalse(testName + " self complement1", ensemble ≡ complement1);
854             assertFalse(testName + " self complement1", other ≡ complement1);
855             assertEquals(testName + " taille 1 failed",
856                     complementElements1.length, complement1.cardinal());
857             boolean compare = compareElts2Array(testName, complement1,
858                     complementElements1);
859             assertTrue(testName + " elts compare 1 failed", compare);
860 
861             Ensemble<String> complement2 = other.complement(ensemble);
862 
863             assertNotNull(testName + " non null complement instance 2 failed",
864                     complement2);
865             assertFalse(testName + " self complement2", ensemble ≡ complement2);
866             assertFalse(testName + " self complement2", other ≡ complement2);
867             assertEquals(testName + " taille 2 failed",
868                     complementElements2.length, complement2.cardinal());
869             compare = compareElts2Array(testName, complement2,
870                     complementElements2);
871             assertTrue(testName + "elts compare 2 failed", compare);
872         }
873     }
874 
875     /**
876      * Test method for {@link ensembles.Ensemble#difference(ensembles.Ensemble)}.
877      */
878     @Test
879     public final void testDifference()
880     {
881         for (int i = 0; i < typesEnsemble.length; i++)
882         {
883             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
884             String otherTypeName = otherType.getSimpleName();
885 
886             String testName = new String(typeName + ".difference("
887                     + otherTypeName + "<E>)");
888             System.out.println(testName);
889 
890             // remplissage ensemble avec singleElements
891             for (String elt : elements1)
892             {
893                 ensemble.ajout(elt);
894             }
895 
896             // remplissage other avec singleElements2
897             Ensemble<String> other = constructEnsemble(testName,
898                     typesEnsemble[i], null);
899             assertNotNull(testName + " other non null instance failed", other);
900 
901             for (String elt : elements2)
902             {
```



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

```
985
986             // Equals sur Ensemble mÃªme contenu ordre diffÃ©rent
987             other.efface();
988             for(String elt : allsingleElementsShuffle)
989             {
990                 other.ajout(elt);
991             }
992
993             // ensemble est toujours sorted car construit avec
994             // allSingleElementsSorted
995             if ((ensemble instanceof EnsembleTri<?>) ∧
996                ¬(other instanceof EnsembleTri<?>))
997             {
998                 assertFalse(testName + " ens identique, ordre diffÃ©rent["
999                         + otherTypeName + "] failed", ensemble.equals(other));
1000            }
1001            else
1002            {
1003                assertEquals(testName + " ens identique, ordre diffÃ©rent["
1004                        + otherTypeName + "] failed", ensemble, other);
1005            }
1006
1007            // Equals sur Ensemble contenu diffÃ©rent
1008            other.ajout("bonjour");
1009            assertFalse(testName + " ens diffÃ©rent failed",
1010                    ensemble.equals(other));
1011        }
1012    }
1013
1014    /**
1015     * Test method for {@link ensembles.Ensemble#hashCode()}.
1016     */
1017    @Test
1018    public final void testHashCode()
1019    {
1020        String testName = new String(typeName + ".hashCode()");
1021        System.out.println(testName);
1022        int hash;
1023        boolean trie = ensemble instanceof EnsembleTri<?>;
1024        if (trie)
1025        {
1026            hash = 1;
1027        }
1028        else
1029        {
1030            hash = 0;
1031        }
1032
1033        // hash code ensemble vide ==
1034        //  0 pour les Ensemble
1035        //  1 pour les EnsembleTri
1036        assertEquals(testName + " hashcode ens vide failed", hash,
1037                ensemble.hashCode());
1038
1039        // hash code ensemble non vide ==
1040        //  somme des hashcode des elts pour les Ensemble
1041        //  comme les collections pour les EnsembleTri
1042        for (String elt : allSingleElements)
1043        {
1044            ensemble.ajout(elt);
1045        }
1046        if (trie)
1047        {
1048            final int prime = 31;
1049            for (String elt : allSingleElementsSorted)
1050            {
1051                hash = (prime * hash) + (elt ≡ null ? 0 : elt.hashCode());
1052            }
1053        }
1054        else
1055        {
1056            for (String elt : allSingleElements)
1057            {
1058                hash += elt.hashCode();
1059            }
1060        }
1061
1062        assertEquals(testName + " hashcode ens non vide failed", hash,
1063                ensemble.hashCode());
1064    }
1065
1066    /**
```



```
1067     * Test method for {@link ensembles.Ensemble#toString()}.
1068     */
1069    @Test
1070    public final void testToString()
1071    {
1072        String testName = new String(typeName + ".toString()");
1073        System.out.println(testName);
1074
1075        ensemble = constructEnsemble(testName, typeEnsemble, listElements);
1076        assertNotNull(testName + " non null instance failed", ensemble);
1077
1078        StringBuilder sb = new StringBuilder();
1079        sb.append("[");
1080        Iterator<String> it = ensemble.iterator();
1081        if (it ≠ null)
1082        {
1083            for (; it.hasNext();)
1084            {
1085                sb.append(it.next().toString());
1086                if (it.hasNext())
1087                {
1088                    sb.append(",");
1089                }
1090            }
1091            sb.append("]");
1092
1093            String expected = sb.toString();
1094
1095            assertEquals(testName, expected, ensemble.toString());
1096        }
1097        else
1098        {
1099            fail(testName + " null iterator");
1100        }
1101    }
1102
1103    /**
1104     * Test method for {@link ensembles.Ensemble#iterator()}.
1105     */
1106    @Test
1107    public final void testIterator()
1108    {
1109        String testName = new String(typeName + ".iterator()");
1110        System.out.println(testName);
1111
1112        Iterator<String> it = null;
1113
1114        // iterator existe
1115        it = ensemble.iterator();
1116        assertNotNull(testName +  " non null instance failed", it);
1117
1118        // iterator sur ens vide n'a pas d'elts Ã  itÃ©rer
1119        assertFalse(testName + " !hasNext() sur ens vide failed", it.hasNext());
1120
1121        // remplissage
1122        for (String elt : allSingleElements)
1123        {
1124            ensemble.ajout(elt);
1125        }
1126
1127        it = ensemble.iterator();
1128
1129        // iterator sur ens rempli
1130        assertTrue(testName + " hashNext() sur ens rempli failed", it.hasNext());
1131
1132        String[] array;
1133        if (ensemble instanceof EnsembleTri<?>)
1134        {
1135            array = allSingleElementsSorted;
1136        }
1137        else
1138        {
1139            array = allSingleElements;
1140        }
1141
1142        // comparaison des elts
1143        for(int i = 0; (i < array.length) ∧ it.hasNext(); i++)
1144        {
1145            assertEquals(testName + "check elt:" + array[i] + " failed",
1146                    array[i], it.next());
1147        }
1148
```

```
1149        // plus l'elts Ã  itÃ©rer
1150        assertFalse(testName + " !hasNext() fin comparaison failed",
1151                it.hasNext());
1152
1153        // retrait des elts avec l'itÃ©rateur
1154        it = ensemble.iterator();
1155        for (int i = 0; (i < array.length) ∧ it.hasNext(); i++)
1156        {
1157            it.next();
1158            it.remove();
1159            assertFalse(testName + " retrait elt:" + array[i] + " failed",
1160                    ensemble.contient(array[i]));
1161        }
1162
1163        // plus l'elts Ã  itÃ©rer
1164        assertFalse(testName + " !hasNext() fin retrait failed", it.hasNext());
1165        assertTrue(testName + " ens vide aprÃ¨s retraits failed",
1166                ensemble.estVide());
1167    }
1168 }
```



```
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.Collections;
14 import java.util.HashMap;
15 import java.util.Iterator;
16 import java.util.List;
17 import java.util.Map;
18
19 import org.junit.After;
20 import org.junit.AfterClass;
21 import org.junit.Before;
22 import org.junit.BeforeClass;
23 import org.junit.Test;
24 import org.junit.runner.RunWith;
25 import org.junit.runners.Parameterized;
26 import org.junit.runners.Parameterized.Parameters;
27
28 import ensembles.Ensemble;
29 import ensembles.EnsembleFactory;
30 import ensembles.EnsembleListe;
31 import ensembles.EnsembleTri;
32
33 /**
34  * Classe de test pour tous les types d'ensembles :
35  * {@link ensembles.EnsembleVector}, {@link ensembles.EnsembleListe},
36  * {@link ensembles.EnsembleTableau}.
37  * Mais aussi pour les mÃ©thodes communes avec les ensemble triÃ©s tels que
38  * {@link ensembles.EnsembleTriVector}, {@link ensembles.EnsembleTriVector2},
39  * {@link ensembles.EnsembleTriListe}, {@link ensembles.EnsembleTriListe2},
40  * {@link ensembles.EnsembleTriTableau}, {@link ensembles.EnsembleTriTableau2}
41  * @author davidroussel
42  */
43 @RunWith(value = Parameterized.class)
44 public class EnsembleListeTest
   {
46     /**
47      * l'ensemble Ã  tester
48      */
49     private Ensemble<String> ensemble;
50
51     /**
52      * Le type d'ensemble Ã  tester.
53      */
54     private Class<? extends Ensemble<String>> typeEnsemble;
55
56     /**
57      * Nom du type d'ensemble Ã  tester
58      */
59     private String typeName;
60
61     /**
62      * Les diffÃ©rentes natures d'ensembles Ã  tester
63      */
64     @SuppressWarnings("unchecked")
65     private static final Class<? extends Ensemble<String>>[] typesEnsemble =
66     (Class<? extends Ensemble<String>>[]) new Class<?>[]
67     {
68         EnsembleListe.class
69     };
70
71     /**
72      * Elements pour remplir l'ensemble : "Lorem ipsum dolor sit amet"
73      */
74     private static final String[] elements1 = new String[] {
75         "Lorem",
76         "ipsum",
77         "sit",
78         "dolor",
79         "amet"
80     };
81
82     /**
```

```
     * 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
     */
    private static final String[] elements = new String[elements1.length
            + elements2.length];

    /**
     * Collection pour contenir les Ã©lements de remplissage
     */
    private ArrayList<String> listElements;

    /**
     * Construit une instance de Ensemble<String> en fonction d'un type
     * d'ensemble Ã  crÃ©er et Ã©ventuellement d'un contenu l'ensemble Ã  mettre en
     * place
     *
     * @param testName le message Ã  rÃ©pÃ©ter dans les assertions en fonction du
     *            test dans lequel est employÃ© cette mÃ©thode
     * @param type le type d'ensemble Ã  crÃ©er
     * @param content le contenu Ã  mettre en place dans le nouvel ensemble, ou
     * bien null si aucun contenu n'est requis.
     * @return un nouvel ensemble du type demandÃ© evt rempli avec le contenu
     * fournit s'il est non null.
     */
    private static Ensemble<String>
    constructEnsemble(String testName,
                      Class<? extends Ensemble<String>> type,
                      Iterable<String> content)
    {
        Ensemble<String> ensemble = null;

        try
        {
            ensemble = EnsembleFactory.<String>getEnsemble(type, content);
        }
        catch (SecurityException e)
        {
            fail(testName +  " constructor security exception");
        }
        catch (NoSuchMethodException e)
        {
            fail(testName +  " constructor not found");
        }
        catch (IllegalArgumentException e)
        {
            fail(testName + " wrong constructor arguments");
        }
        catch (InstantiationException e)
        {
            fail(testName + " instanciation exception");
        }
        catch (IllegalAccessException e)
        {
            fail(testName + " illegal access");
        }
        catch (InvocationTargetException e)
        {
            fail(testName + " invocation exception");
        }

        return ensemble;
    }

    /**
     * Compare les Ã©lÃ©ments d'un ensemble pour vÃ©rifier qu'ils sont tous dans
     * un tableau donnÃ©
     * @param testName le nom du test dans lequel est utilisÃ© cette mÃ©thode
     * @param ensemble l'ensemble dont on doit comparer les Ã©lÃ©ments
     * @param array le tableau utilisÃ© pour vÃ©rifier la prÃ©sence des Ã©lÃ©ments
     * de l'ensemble
     * @return true si tous les Ã©lÃ©ments du tableau sont prÃ©sents dans l'ensemble
     */
    private static boolean compareElts2Array(String testName,
            Ensemble<String> ensemble, String[] array)
    {
        for (String elt : array)
        {
            boolean contenu = ensemble.contient(elt);
            assertTrue(testName + " contient(" + elt + ") failed", contenu);
            if (¬contenu)
            {
                return false;
```

```
247             }
248         }
249         return true;
250     }
251
252     /**
253      * VÃ©rifie qu'un ensemble ne contient qu'un seul exemplaire de chacun
254      * de ses Ã©lÃ©ments
255      * @param testName le nom du test dans lequel est employÃ©e cette mÃ©thode
256      * @param ensemble l'ensemble Ã  tester
257      * @return true si chaque Ã©lÃ©ment de l'ensemble n'existe qu'Ã  un seul
258      * exemplaire.
259      */
260     private static <E> boolean checkCount(String testName, Ensemble<E> ensemble)
261     {
262         Map<E, Integer> wordCount = new HashMap<E, Integer>();
263         for (E elt : ensemble)
264         {
265             if (¬wordCount.containsKey(elt))
266             {
267                 wordCount.put(elt, Integer.valueOf(1));
268             }
269             else
270             {
271                 Integer count = wordCount.get(elt);
272                 count = Integer.valueOf(count.intValue() + 1);
273                 wordCount.put(elt, count);
274             }
275         }
276
277         for (Integer i : wordCount.values())
278         {
279             int countValue = i.intValue();
280             assertEquals(testName + " count check #" + countValue + " failed",
281                     1, countValue);
282             if (countValue ≠ 1)
283             {
284                 return false;
285             }
286         }
287
288         return true;
289     }
290
291     /**
292      * MÃ©lange les Ã©lÃ©ments d'un tableau
293      * @param elements les Ã©lements Ã  mÃ©langer
294      * @return un tableau de mÃªme dimension avec les Ã©lÃ©ments dans un autre
295      * ordre
296      */
297     private static String[] shuffleElements(String[] elements)
298     {
299         List<String> listElements = Arrays.asList(elements);
300
301         Collections.shuffle(listElements);
302
303         String[] result = new String[elements.length];
304         int i = 0;
305         for (String elt : listElements)
306         {
307             result[i++] = elt;
308         }
309
310         return result;
311     }
312
313     /**
314      * ParamÃ¨tres Ã  transmettre au constructeur de la classe de test.
315      *
316      * @return une collection de tableaux d'objet contenant les paramÃ¨tres Ã 
317      *         transmettre au constructeur de la classe de test
318      */
319     @Parameters(name = "{index}:{1}")
320     public static Collection<Object[]> data()
321     {
322         Object[][] data = new Object[typesEnsemble.length][2];
323         for (int i = 0; i < typesEnsemble.length; i++)
324         {
325             data[i][0] = typesEnsemble[i];
326             data[i][1] = typesEnsemble[i].getSimpleName();
327         }
328
```



```
329         return Arrays.asList(data);
330     }
331
332     /**
333      * Constructeur paramÃ¨trÃ© par le type d'ensemble Ã  tester.
334      * LancÃ© pour chaque test
335      * @param typeEnsemble le type d'ensemble Ã  gÃ©nÃ©rer
336      * @param le nom du type d'ensemble Ã  tester (pour le faire apparaÃ®tre
337      * dans le dÃ©roulement des tests).
338      */
339     public EnsembleListeTest(Class<? extends Ensemble<String>> typeEnsemble,
340                              String typeEnsembleName)
341     {
342         this.typeEnsemble = typeEnsemble;
343         typeName = typeEnsembleName;
344     }
345
346     /**
347      * Mise en place avant l'ensemble des tests
348      * @throws java.lang.Exception
349      */
350     @BeforeClass
351     public static void setUpBeforeClass() throws Exception
352     {
353         int j = 0;
354         for (int i = 0; i < elements1.length; i++)
355         {
356             elements[j++] = elements1[i];
357         }
358         for (int i = 0; i < elements2.length; i++)
359         {
360             elements[j++] = elements2[i];
361         }
362         System.out.println("------------------------------------------------");
363         System.out.println("Test des ensembles");
364         System.out.println("------------------------------------------------");
365     }
366
367     /**
368      * Nettoyage aprÃ¨s l'ensemble des tests
369      * @throws java.lang.Exception
370      */
371     @AfterClass
372     public static void tearDownAfterClass() throws Exception
373     {
374         System.out.println("------------------------------------------------");
375         System.out.println("Fin Test des ensembles");
376         System.out.println("------------------------------------------------");
377     }
378
379     /**
380      * Mise en place avant chaque test
381      * @throws java.lang.Exception
382      */
383     @Before
384     public void setUp() throws Exception
385     {
386         ensemble = constructEnsemble("setUp", typeEnsemble, null);
387         assertNotNull("setUp non null ensemble failed", ensemble);
388
389         listElements = new ArrayList<String>();
390         for (String elt : elements)
391         {
392             listElements.add(elt);
393         }
394     }
395
396     /**
397      * Nettoyage aprÃ¨s chaque test
398      * @throws java.lang.Exception
399      */
400     @After
401     public void tearDown() throws Exception
402     {
403         ensemble.efface();
404         ensemble = null;
405         listElements.clear();
406         listElements = null;
407     }
408
409     /**
410      * Test method for {@link ensembles.EnsembleVector#EnsembleVector()} or
```

```
411      * {@link ensembles.EnsembleListe#EnsembleListe()} or
412      * {@link ensembles.EnsembleTableau#EnsembleTableau()}
413      */
414     @Test
415     public final void testDefaultConstructor()
416     {
417         String testName = new String(typeName + "()");
418         System.out.println(testName);
419 
420         ensemble = constructEnsemble(testName, typeEnsemble, null);
421         assertNotNull(testName + " non null instance failed", ensemble);
422 
423         assertEquals(testName + " instance type failed", typeEnsemble,
424                 ensemble.getClass());
425         assertTrue(testName +  " empty instance failed", ensemble.estVide());
426         assertEquals(testName + " instance size failed", 0, ensemble.cardinal());
427     }
428 
429     /**
430      * Test method for {@link ensembles.EnsembleVector#EnsembleVector(Iterable)}
431      * or {@link ensembles.EnsembleListe#EnsembleListe(Iterable)} or
432      * {@link ensembles.EnsembleTableau#EnsembleTableau(Iterable)}
433      */
434     @Test
435     public final void testCopyConstructor()
436     {
437         String testName = new String(typeName + "(Iterable)");
438         System.out.println(testName);
439 
440         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
441         assertNotNull(testName + " non null instance failed", ensemble);
442 
443         assertEquals(testName + " instance type failed", typeEnsemble,
444                 ensemble.getClass());
445         assertFalse(testName +  " not empty instance failed", ensemble.estVide());
446         boolean compare = compareElts2Array(testName, ensemble, allSingleElements);
447         assertTrue(testName + "elts compare failed", compare);
448 
449         // Tous les Ã©lÃ©ments de ensemble doivent se retrouver dans list
450         for (String elt : ensemble)
451         {
452             assertTrue(testName + "check content [" + elt + "] failed",
453                     listElements.contains(elt));
454         }
455 
456         // Tous les Ã©lÃ©ments de l'ensemble n'existent qu'Ã  un seul exemplaire
457         boolean countCheck = EnsembleListeTest.<String>checkCount(testName, ensemble);
458 
459         assertTrue(testName + "after count check failed", countCheck);
460     }
461 
462     /**
463      * Test method for {@link ensembles.Ensemble#ajout(java.lang.Object)}.
464      */
465     @Test
466     public final void testAjout()
467     {
468         String testName = new String(typeName + ".ajout(E)");
469         System.out.println(testName);
470 
471         // Ensemble vide avant remplissage
472         assertEquals(testName + " ensemble vide failed", 0, ensemble.cardinal());
473         int count = 0;
474         for (String elt : elements)
475         {
476             if (¬ensemble.contient(elt))
477             {
478                 count++;
479             }
480             ensemble.ajout(elt);
481         }
482         // Ensemble non vide aprÃ¨s remplissage
483         assertEquals(testName + " ensemble rempli failed", count,
484                 ensemble.cardinal());
485 
486         // Verif taille ensemble
487         boolean countCheck = EnsembleListeTest.<String>checkCount(testName, ensemble);
488         assertTrue(testName + "after count check failed", countCheck);
489 
490         // Comparaison des elts avec allSingleElements
491         boolean compare = compareElts2Array(testName, ensemble, allSingleElements);
492         assertTrue(testName + "elts compare failed", compare);
```



```
493 
494         // Ajout d'un elt null
495         boolean ajoutNull = ensemble.ajout(null);
496         assertFalse(testName + " ajout null is true", ajoutNull);
497     }
498 
499     /**
500      * Test method for {@link ensembles.Ensemble#retrait(java.lang.Object)}.
501      */
502     @Test
503     public final void testRetrait()
504     {
505         String testName = new String(typeName + ".retrait(E)");
506         System.out.println(testName);
507 
508         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
509         assertNotNull(testName + " non null instance failed", ensemble);
510 
511         String[] elementsToRemove = shuffleElements(allSingleElements);
512 
513         for (String elt : elementsToRemove)
514         {
515             ensemble.retrait(elt);
516 
517             assertFalse(testName + "no more contains " + elt + " failed",
518                     ensemble.contient(elt));
519         }
520 
521         assertTrue(testName + " ensemble vide aprÃ¨s retraits failed",
522                 ensemble.estVide());
523     }
524 
525     /**
526      * Test method for {@link ensembles.Ensemble#estVide()}.
527      */
528     @Test
529     public final void testEstVide()
530     {
531         String testName = new String(typeName + ".estVide()");
532         System.out.println(testName);
533 
534         assertTrue(testName + " ensemble vide failed", ensemble.estVide());
535         assertFalse(testName + " ens vide rien Ã  itÃ©rer failed",
536                 ensemble.iterator().hasNext());
537 
538         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
539         assertNotNull(testName + " non null instance failed", ensemble);
540 
541         assertFalse(testName + " ensemble vide failed", ensemble.estVide());
542         assertTrue(testName + " ens non vide iterable failed",
543                 ensemble.iterator().hasNext());
544     }
545 
546     /**
547      * Test method for {@link ensembles.Ensemble#contient(java.lang.Object)}.
548      */
549     @Test
550     public final void testContientENull()
551     {
552         String testName = new String(typeName + ".contient((E)null)");
553         System.out.println(testName);
554         String mot = null;
555 
556         // Contient null sur ensemble vide
557         assertFalse(testName + " ens vide !contient(null) failed",
558                 ensemble.contient(mot));
559 
560         // remplissage ensemble
561         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
562         assertNotNull(testName + " non null instance failed", ensemble);
563         assertEquals(testName + " instance remplie failed",
564                 allSingleElements.length, ensemble.cardinal());
565 
566         // Contient null sur ensemble non vide
567         assertFalse(testName + " ens plein !contient(null) failed",
568                 ensemble.contient((String) null));
569     }
570 
571     /**
572      * Test method for {@link ensembles.Ensemble#contient(java.lang.Object)}.
573      */
574     @Test
```

```
575     public final void testContientE()
576     {
577         String testName = new String(typeName + ".contient(E)");
578         System.out.println(testName);
579         String mot = new String("Bonjour");
580 
581         // Contient mot quelconque sur ensemble vide
582         assertFalse(testName + " ens vide !contient(" + mot + ") failed",
583                 ensemble.contient(mot));
584 
585         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
586         assertNotNull(testName + " non null instance failed", ensemble);
587 
588         // Contient mot quelconque sur ensemble non vide
589         assertFalse(testName + " ens vide contient(" + mot + ") failed",
590                 ensemble.contient(mot));
591 
592         // Contient mots contenus
593         boolean compare = compareElts2Array(testName, ensemble, allSingleElements);
594         assertTrue(testName + " elts compare failed", compare);
595     }
596 
597     /**
598      * Test method for {@link ensembles.Ensemble#contient(ensembles.Ensemble)}.
599      */
600     @Test
601     public final void testContientEnsembleNull()
602     {
603         String testName = new String(typeName + ".contient((Ensemble<E>)null)");
604         System.out.println(testName);
605 
606         // !Contient ensemble null dans ensemble vide
607         assertFalse(testName + "ens vide !contient(null) failed",
608                 ensemble.contient((Ensemble<String>) null));
609 
610         // !Contient ensemble null dans ensemble plein
611         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
612         assertNotNull(testName + " non null instance failed", ensemble);
613         assertEquals(testName + " instance remplie taille failed",
614                 allSingleElements.length, ensemble.cardinal());
615 
616         assertFalse(testName + "ens plein non !contient(null) failed",
617                 ensemble.contient((Ensemble<String>) null));
618     }
619 
620     /**
621      * Test method for {@link ensembles.Ensemble#contient(ensembles.Ensemble)}.
622      */
623     @Test
624     public final void testContientEnsembleOfE()
625     {
626         for (int i = 0; i < typesEnsemble.length; i++)
627         {
628             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
629             String otherTypeName = otherType.getSimpleName();
630 
631             String testName = new String(typeName + ".contient("
632                     + otherTypeName + "<E>)");
633             System.out.println(testName);
634 
635             // sous ensemble vide
636             Ensemble<String> sousEnsemble = constructEnsemble(testName,
637                     typesEnsemble[i], null);
638             assertNotNull(testName + " sousEnsemble non null instance failed",
639                     sousEnsemble);
640 
641             // Contient sous ensemble vide dans ensemble vide
642             assertTrue(testName + " ens vide contient sous ens["
643                     + typesEnsemble[i].getSimpleName() + "] vide failed",
644                     ensemble.contient(sousEnsemble));
645 
646             // remplissage ensemble
647             for (String elt : elements1)
648             {
649                 ensemble.ajout(elt);
650             }
651 
652             // Contient sous ensemble vide dans ensemble non vide
653             assertTrue(testName + " ens plein contient sous ens["
654                     + typesEnsemble[i].getSimpleName() + "] vide failed",
655                     ensemble.contient(sousEnsemble));
656 
```



```
657             // remplissage sous ensemble
658             for (int j = 0; j < (elements1.length / 2); j++)
659             {
660                 sousEnsemble.ajout(elements1[j]);
661             }
662 
663             // Contient sous ensemble non vide ds ens non vide
664             assertTrue(testName + " ens plein contient sous ens["
665                     + typesEnsemble[i].getSimpleName() + "] failed",
666                     ensemble.contient(sousEnsemble));
667 
668             // !Contient sous ensemble non vide non contenu ds ens non vide
669             sousEnsemble.ajout("consectetur");
670             assertFalse(testName + " ens plein !contient sous ens["
671                     + typesEnsemble[i].getSimpleName() + "] failed",
672                     ensemble.contient(sousEnsemble));
673 
674             ensemble.efface();
675         }
676     }
677 
678     /**
679      * Test method for {@link ensembles.Ensemble#efface()}.
680      */
681     @Test
682     public final void testEfface()
683     {
684         String testName = new String(typeName + ".efface()");
685         System.out.println(testName);
686 
687         assertTrue(testName + "ens vide avant effacement failed",
688                 ensemble.estVide());
689 
690         // Effacement ensemble vide
691         ensemble.efface();
692         assertTrue(testName + "ens vide aprÃ¨s effacement failed", ensemble.estVide());
693 
694         // Effacement ensemble non vide
695         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
696         assertNotNull(testName + " non null instance failed", ensemble);
697         assertFalse(testName + "ens non vide aprÃ¨s remplissage failed",
698                 ensemble.estVide());
699         ensemble.efface();
700         assertTrue(testName + "ens vide aprÃ¨s remplissage & effacement failed",
701                 ensemble.estVide());
702     }
703 
704     /**
705      * Test method for {@link ensembles.Ensemble#cardinal()}.
706      */
707     @Test
708     public final void testCardinal()
709     {
710         String testName = new String(typeName + ".cardinal()");
711         System.out.println(testName);
712 
713         assertTrue(testName + " ensemble vide failed", ensemble.estVide());
714         assertEquals(testName + " cardinal 0 sur ensemble vide failed", 0,
715                 ensemble.cardinal());
716 
717         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
718         assertNotNull(testName + " non null instance failed", ensemble);
719 
720         assertFalse(testName + " ensemble non vide failed", ensemble.estVide());
721         assertEquals(testName + " cardinal " + allSingleElements.length
722                 + " sur ensemble rempli failed", allSingleElements.length,
723                 ensemble.cardinal());
724     }
725 
726     /**
727      * Test method for {@link ensembles.Ensemble#union(ensembles.Ensemble)}.
728      */
729     @Test
730     public final void testUnion()
731     {
732         for (int i = 0; i < typesEnsemble.length; i++)
733         {
734             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
735             String otherTypeName = otherType.getSimpleName();
736 
737             String testName = new String(typeName + ".union(" + otherTypeName
738                     + "<E>)");
```

```
739             System.out.println(testName);
740
741             // remplissage ensemble avec singleElements
742             for (String elt : elements1)
743             {
744                 ensemble.ajout(elt);
745             }
746
747             // remplissage other avec singleElements2
748             Ensemble<String> other = constructEnsemble(testName,
749                     typesEnsemble[i], null);
750             assertNotNull(testName + " other instance non null failed", other);
751             for (String elt : elements2)
752             {
753                 other.ajout(elt);
754             }
755
756             Ensemble<String> union = ensemble.union(other);
757
758             assertNotNull(testName + " non null union instance failed", union);
759             assertFalse(testName + " self union", ensemble ≡ union);
760             assertFalse(testName + " self union", other ≡ union);
761             assertEquals(testName + " taille failed",
762                     allSingleElements.length, union.cardinal());
763             boolean compare = compareElts2Array(testName, union,
764                     allSingleElements);
765             assertTrue(testName + " elts compare failed", compare);
766         }
767     }
768
769     /**
770      * Test method for {@link ensembles.Ensemble#intersection(ensembles.Ensemble)}.
771      */
772     @Test
773     public final void testIntersection()
774     {
775         for (int i = 0; i < typesEnsemble.length; i++)
776         {
777             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
778             String otherTypeName = otherType.getSimpleName();
779
780             String testName = new String(typeName + ".intersection("
781                     + otherTypeName + "<E>)");
782             System.out.println(testName);
783
784             // remplissage ensemble avec singleElements
785             for (String elt : elements1)
786             {
787                 ensemble.ajout(elt);
788             }
789
790             // remplissage other avec singleElements2
791             Ensemble<String> other = constructEnsemble(testName,
792                     typesEnsemble[i], null);
793             assertNotNull(testName + " other non null instance failed", other);
794             for (String elt : elements2)
795             {
796                 other.ajout(elt);
797             }
798
799             Ensemble<String> intersection = ensemble.intersection(other);
800
801             assertNotNull(testName + " non null intersection instance failed",
802                     intersection);
803             assertFalse(testName + " self intersection", ensemble ≡ intersection);
804             assertFalse(testName + " self intersection", other ≡ intersection);
805             assertEquals(testName + " taille failed",
806                     commonSingleElements.length, intersection.cardinal());
807             boolean compare = compareElts2Array(testName, intersection,
808                     commonSingleElements);
809             assertTrue(testName + " elts compare failed", compare);
810         }
811     }
812
813     /**
814      * Test method for {@link ensembles.Ensemble#complement(ensembles.Ensemble)}.
815      */
816     @Test
817     public final void testComplement()
818     {
819         for (int i = 0; i < typesEnsemble.length; i++)
820         {
```



```
821             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
822             String otherTypeName = otherType.getSimpleName();
823
824             String testName = new String(typeName + ".complement("
825                     + otherTypeName + "<E>)");
826             System.out.println(testName);
827
828             // remplissage ensemble avec singleElements
829             for (String elt : elements1)
830             {
831                 ensemble.ajout(elt);
832             }
833
834             // remplissage other avec singleElements2
835             Ensemble<String> other = constructEnsemble(testName,
836                     typesEnsemble[i], null);
837             assertNotNull(testName + " other non null instance failed", other);
838             for (String elt : elements2)
839             {
840                 other.ajout(elt);
841             }
842
843             Ensemble<String> complement1 = ensemble.complement(other);
844
845             assertNotNull(testName + " non null complement instance 1 failed",
846                     complement1);
847             assertFalse(testName + " self complement1", ensemble ≡ complement1);
848             assertFalse(testName + " self complement1", other ≡ complement1);
849             assertEquals(testName + " taille 1 failed",
850                     complementElements1.length, complement1.cardinal());
851             boolean compare = compareElts2Array(testName, complement1,
852                     complementElements1);
853             assertTrue(testName + " elts compare 1 failed", compare);
854
855             Ensemble<String> complement2 = other.complement(ensemble);
856
857             assertNotNull(testName + " non null complement instance 2 failed",
858                     complement2);
859             assertFalse(testName + " self complement2", ensemble ≡ complement2);
860             assertFalse(testName + " self complement2", other ≡ complement2);
861             assertEquals(testName + " taille 2 failed",
862                     complementElements2.length, complement2.cardinal());
863             compare = compareElts2Array(testName, complement2,
864                     complementElements2);
865             assertTrue(testName + "elts compare 2 failed", compare);
866         }
867     }
868
869     /**
870      * Test method for {@link ensembles.Ensemble#difference(ensembles.Ensemble)}.
871      */
872     @Test
873     public final void testDifference()
874     {
875         for (int i = 0; i < typesEnsemble.length; i++)
876         {
877             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
878             String otherTypeName = otherType.getSimpleName();
879
880             String testName = new String(typeName + ".difference("
881                     + otherTypeName + "<E>)");
882             System.out.println(testName);
883
884             // remplissage ensemble avec singleElements
885             for (String elt : elements1)
886             {
887                 ensemble.ajout(elt);
888             }
889
890             // remplissage other avec singleElements2
891             Ensemble<String> other = constructEnsemble(testName,
892                     typesEnsemble[i], null);
893             assertNotNull(testName + " other non null instance failed", other);
894
895             for (String elt : elements2)
896             {
897                 other.ajout(elt);
898             }
899
900             Ensemble<String> difference = ensemble.difference(other);
901
902             assertNotNull(testName + " difference non null instance failed",
```

```
                    difference);
            assertFalse(testName + " self difference", ensemble ≡ difference);
            assertFalse(testName + " self difference", other ≡ difference);
            assertEquals(testName + " taille failed", diffSingleElements.length,
                    difference.cardinal());
            boolean compare = compareElts2Array(testName, difference,
                    diffSingleElements);
            assertTrue(testName + " elts compare failed", compare);
        }
    }

    /**
     * Test method for {@link ensembles.Ensemble#typeElements()}.
     */
    @Test
    public final void testTypeElements()
    {
        String testName = new String(typeName + ".typeElements()");
        System.out.println(testName);

        assertNotNull(testName + " non null instance failed", ensemble);

        // type elt sur ensemble vide == null
        assertEquals(testName + " sur ens vide failed", null,
                ensemble.typeElements());

        // type elt sur ensemble non vide == String
        ensemble = constructEnsemble(testName, typeEnsemble, listElements);
        assertNotNull(testName + " non null instance failed", ensemble);
        assertEquals(testName + " sur ens non vide failed", String.class,
                ensemble.typeElements());
    }

    /**
     * Test method for {@link ensembles.Ensemble#equals(java.lang.Object)}.
     */
    @Test
    public final void testEquals()
    {
        String testName = new String(typeName + ".equals(Object)");
        System.out.println(testName);

        // Equals sur null
        assertFalse(testName + " sur null failed", ensemble.equals(null));

        // Equals sur this
        assertTrue(testName + " sur this failed", ensemble.equals(ensemble));

        // Equals sur autre objet
        assertFalse(testName + " sur Object failed",
                ensemble.equals(new Object()));

        // remplissage ensemble
        for (String elt : allSingleElementsSorted)
        {
            ensemble.ajout(elt);
        }

        String[] allsingleElementsShuffle = shuffleElements(allSingleElements);

        for (int i = 0; i < typesEnsemble.length; i++)
        {
            Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
            String otherTypeName = otherType.getSimpleName();

            Ensemble<String> other = constructEnsemble(testName,
                    typesEnsemble[i], null);

            // Equals sur Ensemble mÃªme contenu mÃªme ordre
            assertNotNull(testName + " other non null instance failed", other);
            for(String elt : allSingleElementsSorted)
            {
                other.ajout(elt);
            }
            assertEquals(testName + " ens identique, ordre identique["
                    + otherTypeName + "] failed", ensemble, other);

            // Equals sur Ensemble mÃªme contenu ordre diffÃ©rent
            other.efface();
            for(String elt : allsingleElementsShuffle)
            {
                other.ajout(elt);
```



```
            }

            // ensemble est toujours sorted car construit avec
            // allSingleElementsSorted
            if ((ensemble instanceof EnsembleTri<?>) ∧
               ¬(other instanceof EnsembleTri<?>))
            {
                assertFalse(testName + " ens identique, ordre diffÃ©rent["
                        + otherTypeName + "] failed", ensemble.equals(other));
            }
            else
            {
                assertEquals(testName + " ens identique, ordre diffÃ©rent["
                        + otherTypeName + "] failed", ensemble, other);
            }

            // Equals sur Ensemble contenu diffÃ©rent
            other.ajout("bonjour");
            assertFalse(testName + " ens diffÃ©rent failed",
                    ensemble.equals(other));
        }
    }

    /**
     * Test method for {@link ensembles.Ensemble#hashCode()}.
     */
    @Test
    public final void testHashCode()
    {
        String testName = new String(typeName + ".hashCode()");
        System.out.println(testName);
        int hash;
        boolean trie = ensemble instanceof EnsembleTri<?>;
        if (trie)
        {
            hash = 1;
        }
        else
        {
            hash = 0;
        }

        // hash code ensemble vide ==
        //  0 pour les Ensemble
        //  1 pour les EnsembleTri
        assertEquals(testName + " hashcode ens vide failed", hash,
                ensemble.hashCode());

        // hash code ensemble non vide ==
        //  somme des hashcode des elts pour les Ensemble
        //  comme les collections pour les EnsembleTri
        for (String elt : allSingleElements)
        {
            ensemble.ajout(elt);
        }
        if (trie)
        {
            final int prime = 31;
            for (String elt : allSingleElementsSorted)
            {
                hash = (prime * hash) + (elt ≡ null ? 0 : elt.hashCode());
            }
        }
        else
        {
            for (String elt : allSingleElements)
            {
                hash += elt.hashCode();
            }
        }

        assertEquals(testName + " hashcode ens non vide failed", hash,
                ensemble.hashCode());
    }

    /**
     * Test method for {@link ensembles.Ensemble#toString()}.
     */
    @Test
    public final void testToString()
    {
        String testName = new String(typeName + ".toString()");
```

```
1067        System.out.println(testName);
1068
1069        ensemble = constructEnsemble(testName, typeEnsemble, listElements);
1070        assertNotNull(testName + " non null instance failed", ensemble);
1071
1072        StringBuilder sb = new StringBuilder();
1073        sb.append("[");
1074        Iterator<String> it = ensemble.iterator();
1075        if (it ≠ null)
1076        {
1077            for (; it.hasNext();)
1078            {
1079                sb.append(it.next().toString());
1080                if (it.hasNext())
1081                {
1082                    sb.append(", ");
1083                }
1084            }
1085            sb.append("]");
1086
1087            String expected = sb.toString();
1088
1089            assertEquals(testName, expected, ensemble.toString());
1090        }
1091        else
1092        {
1093            fail(testName + " null iterator");
1094        }
1095    }
1096
1097    /**
1098     * Test method for {@link ensembles.Ensemble#iterator()}.
1099     */
1100    @Test
1101    public final void testIterator()
1102    {
1103        String testName = new String(typeName + ".iterator()");
1104        System.out.println(testName);
1105
1106        Iterator<String> it = null;
1107
1108        // iterator existe
1109        it = ensemble.iterator();
1110        assertNotNull(testName +  " non null instance failed", it);
1111
1112        // iterator sur ens vide n'a pas d'elts Ã  itÃ©rer
1113        assertFalse(testName + " !hasNext() sur ens vide failed", it.hasNext());
1114
1115        // remplissage
1116        for (String elt : allSingleElements)
1117        {
1118            ensemble.ajout(elt);
1119        }
1120
1121        it = ensemble.iterator();
1122
1123        // iterator sur ens rempli
1124        assertTrue(testName + " hashNext() sur ens rempli failed", it.hasNext());
1125
1126        String[] array;
1127        if (ensemble instanceof EnsembleTri<?>)
1128        {
1129            array = allSingleElementsSorted;
1130        }
1131        else
1132        {
1133            array = allSingleElements;
1134        }
1135
1136        // comparaison des elts
1137        for(int i = 0; (i < array.length) ∧ it.hasNext(); i++)
1138        {
1139            assertEquals(testName + "check elt:" + array[i] + " failed",
1140                    array[i], it.next());
1141        }
1142
1143        // plus l'elts Ã  itÃ©rer
1144        assertFalse(testName + " !hasNext() fin comparaison failed",
1145                it.hasNext());
1146
1147        // retrait des elts avec l'itÃ©rateur
1148        it = ensemble.iterator();
```



```
1149        for (int i = 0; (i < array.length) ∧ it.hasNext(); i++)
1150        {
1151            it.next();
1152            it.remove();
1153            assertFalse(testName + " retrait elt:" + array[i] + " failed",
1154                    ensemble.contient(array[i]));
1155        }
1156
1157        // plus l'elts Ã  itÃ©rer
1158        assertFalse(testName + " !hasNext() fin retrait failed", it.hasNext());
1159        assertTrue(testName + " ens vide aprÃ¨s retraits failed",
1160                ensemble.estVide());
1161    }
1162 }
```

```
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.Collections;
14  import java.util.HashMap;
15  import java.util.Iterator;
16  import java.util.List;
17  import java.util.Map;
18
19  import org.junit.After;
20  import org.junit.AfterClass;
21  import org.junit.Before;
22  import org.junit.BeforeClass;
23  import org.junit.Test;
24  import org.junit.runner.RunWith;
25  import org.junit.runners.Parameterized;
26  import org.junit.runners.Parameterized.Parameters;
27
28  import ensembles.Ensemble;
29  import ensembles.EnsembleFactory;
30  import ensembles.EnsembleTableau;
31  import ensembles.EnsembleTri;
32
33  /**
34   * Classe de test pour tous les types d'ensembles :
35   * {@link ensembles.EnsembleVector}, {@link ensembles.EnsembleListe},
36   * {@link ensembles.EnsembleTableau}.
37   * Mais aussi pour les mÃ©thodes communes avec les ensemble triÃ©s tels que
38   * {@link ensembles.EnsembleTriVector}, {@link ensembles.EnsembleTriVector2},
39   * {@link ensembles.EnsembleTriListe}, {@link ensembles.EnsembleTriListe2},
40   * {@link ensembles.EnsembleTriTableau}, {@link ensembles.EnsembleTriTableau2}
41   * @author davidroussel
42   */
43  @RunWith(value = Parameterized.class)
44  public class EnsembleTableauTest
    {
46      /**
47       * l'ensemble Ã  tester
48       */
49      private Ensemble<String> ensemble;
50
51      /**
52       * Le type d'ensemble Ã  tester.
53       */
54      private Class<? extends Ensemble<String>> typeEnsemble;
55
56      /**
57       * Nom du type d'ensemble Ã  tester
58       */
59      private String typeName;
60
61      /**
62       * Les diffÃ©rentes natures d'ensembles Ã  tester
63       */
64      @SuppressWarnings("unchecked")
65      private static final Class<? extends Ensemble<String>>[] typesEnsemble =
66      (Class<? extends Ensemble<String>>[]) new Class<?>[]
67      {
68          EnsembleTableau.class
69      };
70
71      /**
72       * Elements pour remplir l'ensemble : "Lorem ipsum dolor sit amet"
73       */
74      private static final String[] elements1 = new String[] {
75          "Lorem",
76          "ipsum",
77          "sit",
78          "dolor",
79          "amet"
80      };
81
82      /**
```



```
83       * Autres Elements pour remplir un ensemble :
84       * "dolor amet consectetur adipisicing elit"
85       */
86      private static final String[] elements2 = new String[] {
87          "dolor",
88          "amet",
89          "consectetur",
90          "adipisicing",
91          "elit"
92      };
93
94      /**
95       * Elements union de {@value #elements1} et {@link #elements2}
96       */
97      private static final String[] allSingleElements = new String[] {
98          "Lorem",
99          "ipsum",
100         "sit",
101         "dolor",
102         "amet",
103         "consectetur",
104         "adipisicing",
105         "elit"
106     };
107
108     /**
109      * Elements union triÃ©e de {@value #elements1} et
110      * {@link #elements2}
111      */
112     private static final String[] allSingleElementsSorted = new String[] {
113         "Lorem",
114         "adipisicing",
115         "amet",
116         "consectetur",
117         "dolor",
118         "elit",
119         "ipsum",
120         "sit"
121     };
122
123     /**
124      * Elements communs Ã  {@value #elements1} et {@link #elements2}
125      */
126     private static final String[] commonSingleElements = new String[] {
127         "dolor",
128         "amet"
129     };
130
131     /**
132      * Elements du complement de {@value #elements1} et
133      * {@link #elements2}
134      */
135     private static final String[] complementElements1 = new String[] {
136         "Lorem",
137         "ipsum",
138         "sit"
139     };
140
141     /**
142      * Elements du complement de {@value #elements2} et
143      * {@link #elements1}
144      */
145     private static final String[] complementElements2 = new String[] {
146         "consectetur",
147         "adipisicing",
148         "elit"
149     };
150
151     /**
152      * Elements non communs Ã  {@value #elements1} et
153      * {@link #elements2}
154      */
155     private static final String[] diffSingleElements = new String[] {
156         "Lorem",
157         "ipsum",
158         "sit",
159         "consectetur",
160         "adipisicing",
161         "elit"
162     };
163
164     /**
```

```
     * Elements pour remplir l'ensemble avec des doublons pour vÃ©rifier que ceux
     * ci ne seront pas ajoutÃ©s dans les ensembles
     */
    private static final String[] elements = new String[elements1.length
            + elements2.length];

    /**
     * Collection pour contenir les Ã©lements de remplissage
     */
    private ArrayList<String> listElements;

    /**
     * Construit une instance de Ensemble<String> en fonction d'un type
     * d'ensemble Ã  crÃ©er et Ã©ventuellement d'un contenu l'ensemble Ã  mettre en
     * place
     *
     * @param testName le message Ã  rÃ©pÃ©ter dans les assertions en fonction du
     *            test dans lequel est employÃ© cette mÃ©thode
     * @param type le type d'ensemble Ã  crÃ©er
     * @param content le contenu Ã  mettre en place dans le nouvel ensemble, ou
     * bien null si aucun contenu n'est requis.
     * @return un nouvel ensemble du type demandÃ© evt rempli avec le contenu
     * fournit s'il est non null.
     */
    private static Ensemble<String>
    constructEnsemble(String testName,
                      Class<? extends Ensemble<String>> type,
                      Iterable<String> content)
    {
        Ensemble<String> ensemble = null;

        try
        {
            ensemble = EnsembleFactory.<String>getEnsemble(type, content);
        }
        catch (SecurityException e)
        {
            fail(testName +  " constructor security exception");
        }
        catch (NoSuchMethodException e)
        {
            fail(testName +  " constructor not found");
        }
        catch (IllegalArgumentException e)
        {
            fail(testName + " wrong constructor arguments");
        }
        catch (InstantiationException e)
        {
            fail(testName + " instanciation exception");
        }
        catch (IllegalAccessException e)
        {
            fail(testName + " illegal access");
        }
        catch (InvocationTargetException e)
        {
            fail(testName + " invocation exception");
        }

        return ensemble;
    }

    /**
     * Compare les Ã©lÃ©ments d'un ensemble pour vÃ©rifier qu'ils sont tous dans
     * un tableau donnÃ©
     * @param testName le nom du test dans lequel est utilisÃ© cette mÃ©thode
     * @param ensemble l'ensemble dont on doit comparer les Ã©lÃ©ments
     * @param array le tableau utilisÃ© pour vÃ©rifier la prÃ©sence des Ã©lÃ©ments
     * de l'ensemble
     * @return true si tous les Ã©lÃ©ments du tableau sont prÃ©sents dans l'ensemble
     */
    private static boolean compareElts2Array(String testName,
            Ensemble<String> ensemble, String[] array)
    {
        for (String elt : array)
        {
            boolean contenu = ensemble.contient(elt);
            assertTrue(testName + " contient(" + elt + ") failed", contenu);
            if (¬contenu)
            {
                return false;
```



```
            }
        }
        return true;
    }

    /**
     * VÃ©rifie qu'un ensemble ne contient qu'un seul exemplaire de chacun
     * de ses Ã©lÃ©ments
     * @param testName le nom du test dans lequel est employÃ©e cette mÃ©thode
     * @param ensemble l'ensemble Ã  tester
     * @return true si chaque Ã©lÃ©ment de l'ensemble n'existe qu'Ã  un seul
     * exemplaire.
     */
    private static <E> boolean checkCount(String testName, Ensemble<E> ensemble)
    {
        Map<E, Integer> wordCount = new HashMap<E, Integer>();
        for (E elt : ensemble)
        {
            if (¬wordCount.containsKey(elt))
            {
                wordCount.put(elt, Integer.valueOf(1));
            }
            else
            {
                Integer count = wordCount.get(elt);
                count = Integer.valueOf(count.intValue() + 1);
                wordCount.put(elt, count);
            }
        }

        for (Integer i : wordCount.values())
        {
            int countValue = i.intValue();
            assertEquals(testName + " count check #" + countValue + " failed",
                    1, countValue);
            if (countValue ≠ 1)
            {
                return false;
            }
        }

        return true;
    }

    /**
     * MÃ©lange les Ã©lÃ©ments d'un tableau
     * @param elements les Ã©lements Ã  mÃ©langer
     * @return un tableau de mÃªme dimension avec les Ã©lÃ©ments dans un autre
     * ordre
     */
    private static String[] shuffleElements(String[] elements)
    {
        List<String> listElements = Arrays.asList(elements);

        Collections.shuffle(listElements);

        String[] result = new String[elements.length];
        int i = 0;
        for (String elt : listElements)
        {
            result[i++] = elt;
        }

        return result;
    }

    /**
     * ParamÃ¨tres Ã  transmettre au constructeur de la classe de test.
     *
     * @return une collection de tableaux d'objet contenant les paramÃ¨tres Ã 
     *         transmettre au constructeur de la classe de test
     */
    @Parameters(name = "{index}:{1}")
    public static Collection<Object[]> data()
    {
        Object[][] data = new Object[typesEnsemble.length][2];
        for (int i = 0; i < typesEnsemble.length; i++)
        {
            data[i][0] = typesEnsemble[i];
            data[i][1] = typesEnsemble[i].getSimpleName();
        }

```

```
329         return Arrays.asList(data);
330     }
331
332     /**
333      * Constructeur paramÃªtrÃ© par le type d'ensemble Ã  tester.
334      * LancÃ© pour chaque test
335      * @param typeEnsemble le type d'ensemble Ã  gÃ©nÃ©rer
336      * @param le nom du type d'ensemble Ã  tester (pour le faire apparaÃ®tre
337      * dans le dÃ©roulement des tests).
338      */
339     public EnsembleTableauTest(Class<? extends Ensemble<String>> typeEnsemble,
340                                String typeEnsembleName)
341     {
342         this.typeEnsemble = typeEnsemble;
343         typeName = typeEnsembleName;
344     }
345
346     /**
347      * Mise en place avant l'ensemble des tests
348      * @throws java.lang.Exception
349      */
350     @BeforeClass
351     public static void setUpBeforeClass() throws Exception
352     {
353         int j = 0;
354         for (int i = 0; i < elements1.length; i++)
355         {
356             elements[j++] = elements1[i];
357         }
358         for (int i = 0; i < elements2.length; i++)
359         {
360             elements[j++] = elements2[i];
361         }
362         System.out.println("------------------------------------------------");
363         System.out.println("Test des ensembles");
364         System.out.println("------------------------------------------------");
365     }
366
367     /**
368      * Nettoyage aprÃ¨s l'ensemble des tests
369      * @throws java.lang.Exception
370      */
371     @AfterClass
372     public static void tearDownAfterClass() throws Exception
373     {
374         System.out.println("------------------------------------------------");
375         System.out.println("Fin Test des ensembles");
376         System.out.println("------------------------------------------------");
377     }
378
379     /**
380      * Mise en place avant chaque test
381      * @throws java.lang.Exception
382      */
383     @Before
384     public void setUp() throws Exception
385     {
386         ensemble = constructEnsemble("setUp", typeEnsemble, null);
387         assertNotNull("setUp non null ensemble failed", ensemble);
388
389         listElements = new ArrayList<String>();
390         for (String elt : elements)
391         {
392             listElements.add(elt);
393         }
394     }
395
396     /**
397      * Nettoyage aprÃ¨s chaque test
398      * @throws java.lang.Exception
399      */
400     @After
401     public void tearDown() throws Exception
402     {
403         ensemble.efface();
404         ensemble = null;
405         listElements.clear();
406         listElements = null;
407     }
408
409     /**
410      * Test method for {@link ensembles.EnsembleVector#EnsembleVector()} or
```



```
411      * {@link ensembles.EnsembleListe#EnsembleListe()} or
412      * {@link ensembles.EnsembleTableau#EnsembleTableau()}
413      */
414     @Test
415     public final void testDefaultConstructor()
416     {
417         String testName = new String(typeName + "()");
418         System.out.println(testName);
419
420         ensemble = constructEnsemble(testName, typeEnsemble, null);
421         assertNotNull(testName + " non null instance failed", ensemble);
422
423         assertEquals(testName + " instance type failed", typeEnsemble,
424                 ensemble.getClass());
425         assertTrue(testName +  " empty instance failed", ensemble.estVide());
426         assertEquals(testName + " instance size failed", 0, ensemble.cardinal());
427     }
428
429     /**
430      * Test method for {@link ensembles.EnsembleVector#EnsembleVector(Iterable)}
431      * or {@link ensembles.EnsembleListe#EnsembleListe(Iterable)} or
432      * {@link ensembles.EnsembleTableau#EnsembleTableau(Iterable)}
433      */
434     @Test
435     public final void testCopyConstructor()
436     {
437         String testName = new String(typeName + "(Iterable)");
438         System.out.println(testName);
439
440         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
441         assertNotNull(testName + " non null instance failed", ensemble);
442
443         assertEquals(testName + " instance type failed", typeEnsemble,
444                 ensemble.getClass());
445         assertFalse(testName +  " not empty instance failed", ensemble.estVide());
446         boolean compare = compareElts2Array(testName, ensemble, allSingleElements);
447         assertTrue(testName + "elts compare failed", compare);
448
449         // Tous les Ã©lÃ©ments de ensemble doivent se retrouver dans list
450         for (String elt : ensemble)
451         {
452             assertTrue(testName + "check content [" + elt + "] failed",
453                     listElements.contains(elt));
454         }
455
456         // Tous les Ã©lÃ©ments de l'ensemble n'existent qu'Ã  un seul exemplaire
457         boolean countCheck = EnsembleTableauTest.<String>checkCount(testName, ensemble);
458
459         assertTrue(testName + "after count check failed", countCheck);
460     }
461
462     /**
463      * Test method for {@link ensembles.Ensemble#ajout(java.lang.Object)}.
464      */
465     @Test
466     public final void testAjout()
467     {
468         String testName = new String(typeName + ".ajout(E)");
469         System.out.println(testName);
470
471         // Ensemble vide avant remplissage
472         assertEquals(testName + " ensemble vide failed", 0, ensemble.cardinal());
473         int count = 0;
474         for (String elt : elements)
475         {
476             if (¬ensemble.contient(elt))
477             {
478                 count++;
479             }
480             ensemble.ajout(elt);
481         }
482         // Ensemble non vide aprÃ¨s remplissage
483         assertEquals(testName + " ensemble rempli failed", count,
484                 ensemble.cardinal());
485
486         // Verif taille ensemble
487         boolean countCheck = EnsembleTableauTest.<String>checkCount(testName, ensemble);
488         assertTrue(testName + "after count check failed", countCheck);
489
490         // Comparaison des elts avec allSingleElements
491         boolean compare = compareElts2Array(testName, ensemble, allSingleElements);
492         assertTrue(testName + "elts compare failed", compare);
```

```
493
494         // Ajout d'un elt null
495         boolean ajoutNull = ensemble.ajout(null);
496         assertFalse(testName + " ajout null is true", ajoutNull);
497     }
498
499     /**
500      * Test method for {@link ensembles.Ensemble#retrait(java.lang.Object)}.
501      */
502     @Test
503     public final void testRetrait()
504     {
505         String testName = new String(typeName + ".retrait(E)");
506         System.out.println(testName);
507
508         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
509         assertNotNull(testName + " non null instance failed", ensemble);
510
511         String[] elementsToRemove = shuffleElements(allSingleElements);
512
513         for (String elt : elementsToRemove)
514         {
515             ensemble.retrait(elt);
516
517             assertFalse(testName + "no more contains " + elt + " failed",
518                     ensemble.contient(elt));
519         }
520
521         assertTrue(testName + " ensemble vide aprÃ¨s retraits failed",
522                 ensemble.estVide());
523     }
524
525     /**
526      * Test method for {@link ensembles.Ensemble#estVide()}.
527      */
528     @Test
529     public final void testEstVide()
530     {
531         String testName = new String(typeName + ".estVide()");
532         System.out.println(testName);
533
534         assertTrue(testName + " ensemble vide failed", ensemble.estVide());
535         assertFalse(testName + " ens vide rien Ã  itÃ©rer failed",
536                 ensemble.iterator().hasNext());
537
538         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
539         assertNotNull(testName + " non null instance failed", ensemble);
540
541         assertFalse(testName + " ensemble vide failed", ensemble.estVide());
542         assertTrue(testName + " ens non vide iterable failed",
543                 ensemble.iterator().hasNext());
544     }
545
546     /**
547      * Test method for {@link ensembles.Ensemble#contient(java.lang.Object)}.
548      */
549     @Test
550     public final void testContientENull()
551     {
552         String testName = new String(typeName + ".contient((E)null)");
553         System.out.println(testName);
554         String mot = null;
555
556         // Contient null sur ensemble vide
557         assertFalse(testName + " ens vide !contient(null) failed",
558                 ensemble.contient(mot));
559
560         // remplissage ensemble
561         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
562         assertNotNull(testName + " non null instance failed", ensemble);
563         assertEquals(testName + " instance remplie failed",
564                 allSingleElements.length, ensemble.cardinal());
565
566         // Contient null sur ensemble non vide
567         assertFalse(testName + " ens plein !contient(null) failed",
568                 ensemble.contient((String) null));
569     }
570
571     /**
572      * Test method for {@link ensembles.Ensemble#contient(java.lang.Object)}.
573      */
574     @Test
```



```
575     public final void testContientE()
576     {
577         String testName = new String(typeName + ".contient(E)");
578         System.out.println(testName);
579         String mot = new String("Bonjour");
580
581         // Contient mot quelconque sur ensemble vide
582         assertFalse(testName + " ens vide !contient(" + mot + ") failed",
583                 ensemble.contient(mot));
584
585         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
586         assertNotNull(testName + " non null instance failed", ensemble);
587
588         // Contient mot quelconque sur ensemble non vide
589         assertFalse(testName + " ens vide contient(" + mot + ") failed",
590                 ensemble.contient(mot));
591
592         // Contient mots contenus
593         boolean compare = compareElts2Array(testName, ensemble, allSingleElements);
594         assertTrue(testName + " elts compare failed", compare);
595     }
596
597     /**
598      * Test method for {@link ensembles.Ensemble#contient(ensembles.Ensemble)}.
599      */
600     @Test
601     public final void testContientEnsembleNull()
602     {
603         String testName = new String(typeName + ".contient((Ensemble<E>)null)");
604         System.out.println(testName);
605
606         // !Contient ensemble null dans ensemble vide
607         assertFalse(testName + "ens vide !contient(null) failed",
608                 ensemble.contient((Ensemble<String>) null));
609
610         // !Contient ensemble null dans ensemble plein
611         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
612         assertNotNull(testName + " non null instance failed", ensemble);
613         assertEquals(testName + " instance remplie taille failed",
614                 allSingleElements.length, ensemble.cardinal());
615
616         assertFalse(testName + "ens plein non !contient(null) failed",
617                 ensemble.contient((Ensemble<String>) null));
618     }
619
620     /**
621      * Test method for {@link ensembles.Ensemble#contient(ensembles.Ensemble)}.
622      */
623     @Test
624     public final void testContientEnsembleOfE()
625     {
626         for (int i = 0; i < typesEnsemble.length; i++)
627         {
628             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
629             String otherTypeName = otherType.getSimpleName();
630
631             String testName = new String(typeName + ".contient("
632                     + otherTypeName + "<E>)");
633             System.out.println(testName);
634
635             // sous ensemble vide
636             Ensemble<String> sousEnsemble = constructEnsemble(testName,
637                     typesEnsemble[i], null);
638             assertNotNull(testName + " sousEnsemble non null instance failed",
639                     sousEnsemble);
640
641             // Contient sous ensemble vide dans ensemble vide
642             assertTrue(testName + " ens vide contient sous ens["
643                     + typesEnsemble[i].getSimpleName() + "] vide failed",
644                     ensemble.contient(sousEnsemble));
645
646             // remplissage ensemble
647             for (String elt : elements1)
648             {
649                 ensemble.ajout(elt);
650             }
651
652             // Contient sous ensemble vide dans ensemble non vide
653             assertTrue(testName + " ens plein contient sous ens["
654                     + typesEnsemble[i].getSimpleName() + "] vide failed",
655                     ensemble.contient(sousEnsemble));
656
```

```
657             // remplissage sous ensemble
658             for (int j = 0; j < (elements1.length / 2); j++)
659             {
660                 sousEnsemble.ajout(elements1[j]);
661             }
662
663             // Contient sous ensemble non vide ds ens non vide
664             assertTrue(testName + " ens plein contient sous ens["
665                     + typesEnsemble[i].getSimpleName() + "] failed",
666                     ensemble.contient(sousEnsemble));
667
668             // !Contient sous ensemble non vide non contenu ds ens non vide
669             sousEnsemble.ajout("consectetur");
670             assertFalse(testName + " ens plein !contient sous ens["
671                     + typesEnsemble[i].getSimpleName() + "] failed",
672                     ensemble.contient(sousEnsemble));
673
674             ensemble.efface();
675         }
676     }
677
678     /**
679      * Test method for {@link ensembles.Ensemble#efface()}.
680      */
681     @Test
682     public final void testEfface()
683     {
684         String testName = new String(typeName + ".efface()");
685         System.out.println(testName);
686
687         assertTrue(testName + "ens vide avant effacement failed",
688                 ensemble.estVide());
689
690         // Effacement ensemble vide
691         ensemble.efface();
692         assertTrue(testName + "ens vide aprÃ¨s effacement failed", ensemble.estVide());
693
694         // Effacement ensemble non vide
695         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
696         assertNotNull(testName + " non null instance failed", ensemble);
697         assertFalse(testName + "ens non vide aprÃ¨s remplissage failed",
698                 ensemble.estVide());
699         ensemble.efface();
700         assertTrue(testName + "ens vide aprÃ¨s remplissage & effacement failed",
701                 ensemble.estVide());
702     }
703
704     /**
705      * Test method for {@link ensembles.Ensemble#cardinal()}.
706      */
707     @Test
708     public final void testCardinal()
709     {
710         String testName = new String(typeName + ".cardinal()");
711         System.out.println(testName);
712
713         assertTrue(testName + " ensemble vide failed", ensemble.estVide());
714         assertEquals(testName + " cardinal 0 sur ensemble vide failed", 0,
715                 ensemble.cardinal());
716
717         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
718         assertNotNull(testName + " non null instance failed", ensemble);
719
720         assertFalse(testName + " ensemble non vide failed", ensemble.estVide());
721         assertEquals(testName + " cardinal " + allSingleElements.length
722                 + " sur ensemble rempli failed", allSingleElements.length,
723                 ensemble.cardinal());
724     }
725
726     /**
727      * Test method for {@link ensembles.Ensemble#union(ensembles.Ensemble)}.
728      */
729     @Test
730     public final void testUnion()
731     {
732         for (int i = 0; i < typesEnsemble.length; i++)
733         {
734             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
735             String otherTypeName = otherType.getSimpleName();
736
737             String testName = new String(typeName + ".union(" + otherTypeName
738                     + "<E>)");
```



```
739             System.out.println(testName);
740
741             // remplissage ensemble avec singleElements
742             for (String elt : elements1)
743             {
744                 ensemble.ajout(elt);
745             }
746
747             // remplissage other avec singleElements2
748             Ensemble<String> other = constructEnsemble(testName,
749                     typesEnsemble[i], null);
750             assertNotNull(testName + " other instance non null failed", other);
751             for (String elt : elements2)
752             {
753                 other.ajout(elt);
754             }
755
756             Ensemble<String> union = ensemble.union(other);
757
758             assertNotNull(testName + " non null union instance failed", union);
759             assertFalse(testName + " self union", ensemble ≡ union);
760             assertFalse(testName + " self union", other ≡ union);
761             assertEquals(testName + " taille failed",
762                     allSingleElements.length, union.cardinal());
763             boolean compare = compareElts2Array(testName, union,
764                     allSingleElements);
765             assertTrue(testName + " elts compare failed", compare);
766         }
767     }
768
769     /**
770      * Test method for {@link ensembles.Ensemble#intersection(ensembles.Ensemble)}.
771      */
772     @Test
773     public final void testIntersection()
774     {
775         for (int i = 0; i < typesEnsemble.length; i++)
776         {
777             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
778             String otherTypeName = otherType.getSimpleName();
779
780             String testName = new String(typeName + ".intersection("
781                     + otherTypeName + "<E>)");
782             System.out.println(testName);
783
784             // remplissage ensemble avec singleElements
785             for (String elt : elements1)
786             {
787                 ensemble.ajout(elt);
788             }
789
790             // remplissage other avec singleElements2
791             Ensemble<String> other = constructEnsemble(testName,
792                     typesEnsemble[i], null);
793             assertNotNull(testName + " other non null instance failed", other);
794             for (String elt : elements2)
795             {
796                 other.ajout(elt);
797             }
798
799             Ensemble<String> intersection = ensemble.intersection(other);
800
801             assertNotNull(testName + " non null intersection instance failed",
802                     intersection);
803             assertFalse(testName + " self intersection", ensemble ≡ intersection);
804             assertFalse(testName + " self intersection", other ≡ intersection);
805             assertEquals(testName + " taille failed",
806                     commonSingleElements.length, intersection.cardinal());
807             boolean compare = compareElts2Array(testName, intersection,
808                     commonSingleElements);
809             assertTrue(testName + " elts compare failed", compare);
810         }
811     }
812
813     /**
814      * Test method for {@link ensembles.Ensemble#complement(ensembles.Ensemble)}.
815      */
816     @Test
817     public final void testComplement()
818     {
819         for (int i = 0; i < typesEnsemble.length; i++)
820         {
```

```
821             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
822             String otherTypeName = otherType.getSimpleName();
823
824             String testName = new String(typeName + ".complement("
825                     + otherTypeName + "<E>)");
826             System.out.println(testName);
827
828             // remplissage ensemble avec singleElements
829             for (String elt : elements1)
830             {
831                 ensemble.ajout(elt);
832             }
833
834             // remplissage other avec singleElements2
835             Ensemble<String> other = constructEnsemble(testName,
836                     typesEnsemble[i], null);
837             assertNotNull(testName + " other non null instance failed", other);
838             for (String elt : elements2)
839             {
840                 other.ajout(elt);
841             }
842
843             Ensemble<String> complement1 = ensemble.complement(other);
844
845             assertNotNull(testName + " non null complement instance 1 failed",
846                     complement1);
847             assertFalse(testName + " self complement1", ensemble ≡ complement1);
848             assertFalse(testName + " self complement1", other ≡ complement1);
849             assertEquals(testName + " taille 1 failed",
850                     complementElements1.length, complement1.cardinal());
851             boolean compare = compareElts2Array(testName, complement1,
852                     complementElements1);
853             assertTrue(testName + " elts compare 1 failed", compare);
854
855             Ensemble<String> complement2 = other.complement(ensemble);
856
857             assertNotNull(testName + " non null complement instance 2 failed",
858                     complement2);
859             assertFalse(testName + " self complement2", ensemble ≡ complement2);
860             assertFalse(testName + " self complement2", other ≡ complement2);
861             assertEquals(testName + " taille 2 failed",
862                     complementElements2.length, complement2.cardinal());
863             compare = compareElts2Array(testName, complement2,
864                     complementElements2);
865             assertTrue(testName + "elts compare 2 failed", compare);
866         }
867     }
868
869     /**
870      * Test method for {@link ensembles.Ensemble#difference(ensembles.Ensemble)}.
871      */
872     @Test
873     public final void testDifference()
874     {
875         for (int i = 0; i < typesEnsemble.length; i++)
876         {
877             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
878             String otherTypeName = otherType.getSimpleName();
879
880             String testName = new String(typeName + ".difference("
881                     + otherTypeName + "<E>)");
882             System.out.println(testName);
883
884             // remplissage ensemble avec singleElements
885             for (String elt : elements1)
886             {
887                 ensemble.ajout(elt);
888             }
889
890             // remplissage other avec singleElements2
891             Ensemble<String> other = constructEnsemble(testName,
892                     typesEnsemble[i], null);
893             assertNotNull(testName + " other non null instance failed", other);
894
895             for (String elt : elements2)
896             {
897                 other.ajout(elt);
898             }
899
900             Ensemble<String> difference = ensemble.difference(other);
901
902             assertNotNull(testName + " difference non null instance failed",
```



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

```
            }

            // ensemble est toujours sorted car construit avec
            // allSingleElementsSorted
            if ((ensemble instanceof EnsembleTri<?>) ∧
               ¬(other instanceof EnsembleTri<?>))
            {
                assertFalse(testName + " ens identique, ordre diffÃ©rent["
                        + otherTypeName + "] failed", ensemble.equals(other));
            }
            else
            {
                assertEquals(testName + " ens identique, ordre diffÃ©rent["
                        + otherTypeName + "] failed", ensemble, other);
            }

            // Equals sur Ensemble contenu diffÃ©rent
            other.ajout("bonjour");
            assertFalse(testName + " ens diffÃ©rent failed",
                    ensemble.equals(other));
        }
    }

    /**
     * Test method for {@link ensembles.Ensemble#hashCode()}.
     */
    @Test
    public final void testHashCode()
    {
        String testName = new String(typeName + ".hashCode()");
        System.out.println(testName);
        int hash;
        boolean trie = ensemble instanceof EnsembleTri<?>;
        if (trie)
        {
            hash = 1;
        }
        else
        {
            hash = 0;
        }

        // hash code ensemble vide ==
        //  0 pour les Ensemble
        //  1 pour les EnsembleTri
        assertEquals(testName + " hashcode ens vide failed", hash,
                ensemble.hashCode());

        // hash code ensemble non vide ==
        //  somme des hashcode des elts pour les Ensemble
        //  comme les collections pour les EnsembleTri
        for (String elt : allSingleElements)
        {
            ensemble.ajout(elt);
        }
        if (trie)
        {
            final int prime = 31;
            for (String elt : allSingleElementsSorted)
            {
                hash = (prime * hash) + (elt ≡ null ? 0 : elt.hashCode());
            }
        }
        else
        {
            for (String elt : allSingleElements)
            {
                hash += elt.hashCode();
            }
        }

        assertEquals(testName + " hashcode ens non vide failed", hash,
                ensemble.hashCode());
    }

    /**
     * Test method for {@link ensembles.Ensemble#toString()}.
     */
    @Test
    public final void testToString()
    {
        String testName = new String(typeName + ".toString()");
```



```
        System.out.println(testName);

        ensemble = constructEnsemble(testName, typeEnsemble, listElements);
        assertNotNull(testName + " non null instance failed", ensemble);

        StringBuilder sb = new StringBuilder();
        sb.append("[");
        Iterator<String> it = ensemble.iterator();
        if (it ≠ null)
        {
            for (; it.hasNext();)
            {
                sb.append(it.next().toString());
                if (it.hasNext())
                {
                    sb.append(",");
                }
            }
            sb.append("]");

            String expected = sb.toString();

            assertEquals(testName, expected, ensemble.toString());
        }
        else
        {
            fail(testName + " null iterator");
        }
    }

    /**
     * Test method for {@link ensembles.Ensemble#iterator()}.
     */
    @Test
    public final void testIterator()
    {
        String testName = new String(typeName + ".iterator()");
        System.out.println(testName);

        Iterator<String> it = null;

        // iterator existe
        it = ensemble.iterator();
        assertNotNull(testName +  " non null instance failed", it);

        // iterator sur ens vide n'a pas d'elts Ã  itÃ©rer
        assertFalse(testName + " !hasNext() sur ens vide failed", it.hasNext());

        // remplissage
        for (String elt : allSingleElements)
        {
            ensemble.ajout(elt);
        }

        it = ensemble.iterator();

        // iterator sur ens rempli
        assertTrue(testName + " hashNext() sur ens rempli failed", it.hasNext());

        String[] array;
        if (ensemble instanceof EnsembleTri<?>)
        {
            array = allSingleElementsSorted;
        }
        else
        {
            array = allSingleElements;
        }

        // comparaison des elts
        for(int i = 0; (i < array.length) ∧ it.hasNext(); i++)
        {
            assertEquals(testName + "check elt:" + array[i] + " failed",
                    array[i], it.next());
        }

        // plus l'elts Ã  itÃ©rer
        assertFalse(testName + " !hasNext() fin comparaison failed",
                it.hasNext());

        // retrait des elts avec l'itÃ©rateur
        it = ensemble.iterator();
```

```
1149            for (int i = 0; (i < array.length) ∧ it.hasNext(); i++)
1150            {
1151                it.next();
1152                it.remove();
1153                assertFalse(testName + " retrait elt:" + array[i] + " failed",
1154                        ensemble.contient(array[i]));
1155            }
1156
1157            // plus l'elts Ã  itÃ©rer
1158            assertFalse(testName + " !hasNext() fin retrait failed", it.hasNext());
1159            assertTrue(testName + " ens vide aprÃ¨s retraits failed",
1160                    ensemble.estVide());
1161        }
1162    }
```



```
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.Collections;
14  import java.util.HashMap;
15  import java.util.Iterator;
16  import java.util.List;
17  import java.util.Map;
18
19  import org.junit.After;
20  import org.junit.AfterClass;
21  import org.junit.Before;
22  import org.junit.BeforeClass;
23  import org.junit.Test;
24  import org.junit.runner.RunWith;
25  import org.junit.runners.Parameterized;
26  import org.junit.runners.Parameterized.Parameters;
27
28  import ensembles.Ensemble;
29  import ensembles.EnsembleFactory;
30  import ensembles.EnsembleTri;
31  import ensembles.EnsembleVector;
32
33  /**
34   * Classe de test pour tous les types d'ensembles :
35   * {@link ensembles.EnsembleVector}, {@link ensembles.EnsembleListe},
36   * {@link ensembles.EnsembleTableau}.
37   * Mais aussi pour les mÃ©thodes communes avec les ensemble triÃ©s tels que
38   * {@link ensembles.EnsembleTriVector}, {@link ensembles.EnsembleTriVector2},
39   * {@link ensembles.EnsembleTriListe}, {@link ensembles.EnsembleTriListe2},
40   * {@link ensembles.EnsembleTriTableau}, {@link ensembles.EnsembleTriTableau2}
41   * @author davidroussel
42   */
43  @RunWith(value = Parameterized.class)
44  public class EnsembleVectorTest
    {
46      /**
47       * l'ensemble Ã  tester
48       */
49      private Ensemble<String> ensemble;
50
51      /**
52       * Le type d'ensemble Ã  tester.
53       */
54      private Class<? extends Ensemble<String>> typeEnsemble;
55
56      /**
57       * Nom du type d'ensemble Ã  tester
58       */
59      private String typeName;
60
61      /**
62       * Les diffÃ©rentes natures d'ensembles Ã  tester
63       */
64      @SuppressWarnings("unchecked")
65      private static final Class<? extends Ensemble<String>>[] typesEnsemble =
66      (Class<? extends Ensemble<String>>[]) new Class<?>[]
67      {
68          EnsembleVector.class
69      };
70
71      /**
72       * Elements pour remplir l'ensemble : "Lorem ipsum dolor sit amet"
73       */
74      private static final String[] elements1 = new String[] {
75          "Lorem",
76          "ipsum",
77          "sit",
78          "dolor",
79          "amet"
80      };
81
82      /**
```

```
     * 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
     */
    private static final String[] elements = new String[elements1.length
            + elements2.length];

    /**
     * Collection pour contenir les Ã©lements de remplissage
     */
    private ArrayList<String> listElements;

    /**
     * Construit une instance de Ensemble<String> en fonction d'un type
     * d'ensemble Ã  crÃ©er et Ã©ventuellement d'un contenu l'ensemble Ã  mettre en
     * place
     *
     * @param testName le message Ã  rÃ©pÃ©ter dans les assertions en fonction du
     *            test dans lequel est employÃ© cette mÃ©thode
     * @param type le type d'ensemble Ã  crÃ©er
     * @param content le contenu Ã  mettre en place dans le nouvel ensemble, ou
     * bien null si aucun contenu n'est requis.
     * @return un nouvel ensemble du type demandÃ© evt rempli avec le contenu
     * fournit s'il est non null.
     */
    private static Ensemble<String>
    constructEnsemble(String testName,
                      Class<? extends Ensemble<String>> type,
                      Iterable<String> content)
    {
        Ensemble<String> ensemble = null;

        try
        {
            ensemble = EnsembleFactory.<String>getEnsemble(type, content);
        }
        catch (SecurityException e)
        {
            fail(testName +  " constructor security exception");
        }
        catch (NoSuchMethodException e)
        {
            fail(testName +  " constructor not found");
        }
        catch (IllegalArgumentException e)
        {
            fail(testName + " wrong constructor arguments");
        }
        catch (InstantiationException e)
        {
            fail(testName + " instanciation exception");
        }
        catch (IllegalAccessException e)
        {
            fail(testName + " illegal access");
        }
        catch (InvocationTargetException e)
        {
            fail(testName + " invocation exception");
        }

        return ensemble;
    }

    /**
     * Compare les Ã©lÃ©ments d'un ensemble pour vÃ©rifier qu'ils sont tous dans
     * un tableau donnÃ©
     * @param testName le nom du test dans lequel est utilisÃ© cette mÃ©thode
     * @param ensemble l'ensemble dont on doit comparer les Ã©lÃ©ments
     * @param array le tableau utilisÃ© pour vÃ©rifier la prÃ©sence des Ã©lÃ©ments
     * de l'ensemble
     * @return true si tous les Ã©lÃ©ments du tableau sont prÃ©sents dans l'ensemble
     */
    private static boolean compareElts2Array(String testName,
            Ensemble<String> ensemble, String[] array)
    {
        for (String elt : array)
        {
            boolean contenu = ensemble.contient(elt);
            assertTrue(testName + " contient(" + elt + ") failed", contenu);
            if (¬contenu)
            {
                return false;
```

```
247             }
248         }
249         return true;
250     }
251
252     /**
253      * VÃ©rifie qu'un ensemble ne contient qu'un seul exemplaire de chacun
254      * de ses Ã©lÃ©ments
255      * @param testName le nom du test dans lequel est employÃ©e cette mÃ©thode
256      * @param ensemble l'ensemble Ã  tester
257      * @return true si chaque Ã©lÃ©ment de l'ensemble n'existe qu'Ã  un seul
258      * exemplaire.
259      */
260     private static <E> boolean checkCount(String testName, Ensemble<E> ensemble)
261     {
262         Map<E, Integer> wordCount = new HashMap<E, Integer>();
263         for (E elt : ensemble)
264         {
265             if (¬wordCount.containsKey(elt))
266             {
267                 wordCount.put(elt, Integer.valueOf(1));
268             }
269             else
270             {
271                 Integer count = wordCount.get(elt);
272                 count = Integer.valueOf(count.intValue() + 1);
273                 wordCount.put(elt, count);
274             }
275         }
276
277         for (Integer i : wordCount.values())
278         {
279             int countValue = i.intValue();
280             assertEquals(testName + " count check #" + countValue + " failed",
281                     1, countValue);
282             if (countValue ≠ 1)
283             {
284                 return false;
285             }
286         }
287
288         return true;
289     }
290
291     /**
292      * MÃ©lange les Ã©lÃ©ments d'un tableau
293      * @param elements les Ã©lements Ã  mÃ©langer
294      * @return un tableau de mÃªme dimension avec les Ã©lÃ©ments dans un autre
295      * ordre
296      */
297     private static String[] shuffleElements(String[] elements)
298     {
299         List<String> listElements = Arrays.asList(elements);
300
301         Collections.shuffle(listElements);
302
303         String[] result = new String[elements.length];
304         int i = 0;
305         for (String elt : listElements)
306         {
307             result[i++] = elt;
308         }
309
310         return result;
311     }
312
313     /**
314      * ParamÃ¨tres Ã  transmettre au constructeur de la classe de test.
315      *
316      * @return une collection de tableaux d'objet contenant les paramÃ¨tres Ã 
317      *         transmettre au constructeur de la classe de test
318      */
319     @Parameters(name = "{index}:{1}")
320     public static Collection<Object[]> data()
321     {
322         Object[][] data = new Object[typesEnsemble.length][2];
323         for (int i = 0; i < typesEnsemble.length; i++)
324         {
325             data[i][0] = typesEnsemble[i];
326             data[i][1] = typesEnsemble[i].getSimpleName();
327         }
328
```



```
329         return Arrays.asList(data);
330     }
331
332     /**
333      * Constructeur paramÃ¨trÃ© par le type d'ensemble Ã  tester.
334      * LancÃ© pour chaque test
335      * @param typeEnsemble le type d'ensemble Ã  gÃ©nÃ©rer
336      * @param le nom du type d'ensemble Ã  tester (pour le faire apparaÃ®tre
337      * dans le dÃ©roulement des tests).
338      */
339     public EnsembleVectorTest(Class<? extends Ensemble<String>> typeEnsemble,
340                               String typeEnsembleName)
341     {
342         this.typeEnsemble = typeEnsemble;
343         typeName = typeEnsembleName;
344     }
345
346     /**
347      * Mise en place avant l'ensemble des tests
348      * @throws java.lang.Exception
349      */
350     @BeforeClass
351     public static void setUpBeforeClass() throws Exception
352     {
353         int j = 0;
354         for (int i = 0; i < elements1.length; i++)
355         {
356             elements[j++] = elements1[i];
357         }
358         for (int i = 0; i < elements2.length; i++)
359         {
360             elements[j++] = elements2[i];
361         }
362         System.out.println("------------------------------------------------");
363         System.out.println("Test des ensembles");
364         System.out.println("------------------------------------------------");
365     }
366
367     /**
368      * Nettoyage aprÃ¨s l'ensemble des tests
369      * @throws java.lang.Exception
370      */
371     @AfterClass
372     public static void tearDownAfterClass() throws Exception
373     {
374         System.out.println("------------------------------------------------");
375         System.out.println("Fin Test des ensembles");
376         System.out.println("------------------------------------------------");
377     }
378
379     /**
380      * Mise en place avant chaque test
381      * @throws java.lang.Exception
382      */
383     @Before
384     public void setUp() throws Exception
385     {
386         ensemble = constructEnsemble("setUp", typeEnsemble, null);
387         assertNotNull("setUp non null ensemble failed", ensemble);
388
389         listElements = new ArrayList<String>();
390         for (String elt : elements)
391         {
392             listElements.add(elt);
393         }
394     }
395
396     /**
397      * Nettoyage aprÃ¨s chaque test
398      * @throws java.lang.Exception
399      */
400     @After
401     public void tearDown() throws Exception
402     {
403         ensemble.efface();
404         ensemble = null;
405         listElements.clear();
406         listElements = null;
407     }
408
409     /**
410      * Test method for {@link ensembles.EnsembleVector#EnsembleVector()} or
```

```
411      * {@link ensembles.EnsembleListe#EnsembleListe()} or
412      * {@link ensembles.EnsembleTableau#EnsembleTableau()}
413      */
414     @Test
415     public final void testDefaultConstructor()
416     {
417         String testName = new String(typeName + "()");
418         System.out.println(testName);
419 
420         ensemble = constructEnsemble(testName, typeEnsemble, null);
421         assertNotNull(testName + " non null instance failed", ensemble);
422 
423         assertEquals(testName + " instance type failed", typeEnsemble,
424                 ensemble.getClass());
425         assertTrue(testName +  " empty instance failed", ensemble.estVide());
426         assertEquals(testName + " instance size failed", 0, ensemble.cardinal());
427     }
428 
429     /**
430      * Test method for {@link ensembles.EnsembleVector#EnsembleVector(Iterable)}
431      * or {@link ensembles.EnsembleListe#EnsembleListe(Iterable)} or
432      * {@link ensembles.EnsembleTableau#EnsembleTableau(Iterable)}
433      */
434     @Test
435     public final void testCopyConstructor()
436     {
437         String testName = new String(typeName + "(Iterable)");
438         System.out.println(testName);
439 
440         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
441         assertNotNull(testName + " non null instance failed", ensemble);
442 
443         assertEquals(testName + " instance type failed", typeEnsemble,
444                 ensemble.getClass());
445         assertFalse(testName +  " not empty instance failed", ensemble.estVide());
446         boolean compare = compareElts2Array(testName, ensemble, allSingleElements);
447         assertTrue(testName + "elts compare failed", compare);
448 
449         // Tous les Ã©lÃ©ments de ensemble doivent se retrouver dans list
450         for (String elt : ensemble)
451         {
452             assertTrue(testName + "check content [" + elt + "] failed",
453                     listElements.contains(elt));
454         }
455 
456         // Tous les Ã©lÃ©ments de l'ensemble n'existent qu'Ã  un seul exemplaire
457         boolean countCheck = EnsembleVectorTest.<String>checkCount(testName, ensemble);
458 
459         assertTrue(testName + "after count check failed", countCheck);
460     }
461 
462     /**
463      * Test method for {@link ensembles.Ensemble#ajout(java.lang.Object)}.
464      */
465     @Test
466     public final void testAjout()
467     {
468         String testName = new String(typeName + ".ajout(E)");
469         System.out.println(testName);
470 
471         // Ensemble vide avant remplissage
472         assertEquals(testName + " ensemble vide failed", 0, ensemble.cardinal());
473         int count = 0;
474         for (String elt : elements)
475         {
476             if (¬ensemble.contient(elt))
477             {
478                 count++;
479             }
480             ensemble.ajout(elt);
481         }
482         // Ensemble non vide aprÃ¨s remplissage
483         assertEquals(testName + " ensemble rempli failed", count,
484                 ensemble.cardinal());
485 
486         // Verif taille ensemble
487         boolean countCheck = EnsembleVectorTest.<String>checkCount(testName, ensemble);
488         assertTrue(testName + "after count check failed", countCheck);
489 
490         // Comparaison des elts avec allSingleElements
491         boolean compare = compareElts2Array(testName, ensemble, allSingleElements);
492         assertTrue(testName + "elts compare failed", compare);
```



```
493 
494         // Ajout d'un elt null
495         boolean ajoutNull = ensemble.ajout(null);
496         assertFalse(testName + " ajout null is true", ajoutNull);
497     }
498 
499     /**
500      * Test method for {@link ensembles.Ensemble#retrait(java.lang.Object)}.
501      */
502     @Test
503     public final void testRetrait()
504     {
505         String testName = new String(typeName + ".retrait(E)");
506         System.out.println(testName);
507 
508         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
509         assertNotNull(testName + " non null instance failed", ensemble);
510 
511         String[] elementsToRemove = shuffleElements(allSingleElements);
512 
513         for (String elt : elementsToRemove)
514         {
515             ensemble.retrait(elt);
516 
517             assertFalse(testName + "no more contains " + elt + " failed",
518                     ensemble.contient(elt));
519         }
520 
521         assertTrue(testName + " ensemble vide aprÃ¨s retraits failed",
522                 ensemble.estVide());
523     }
524 
525     /**
526      * Test method for {@link ensembles.Ensemble#estVide()}.
527      */
528     @Test
529     public final void testEstVide()
530     {
531         String testName = new String(typeName + ".estVide()");
532         System.out.println(testName);
533 
534         assertTrue(testName + " ensemble vide failed", ensemble.estVide());
535         assertFalse(testName + " ens vide rien Ã  itÃ©rer failed",
536                 ensemble.iterator().hasNext());
537 
538         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
539         assertNotNull(testName + " non null instance failed", ensemble);
540 
541         assertFalse(testName + " ensemble vide failed", ensemble.estVide());
542         assertTrue(testName + " ens non vide iterable failed",
543                 ensemble.iterator().hasNext());
544     }
545 
546     /**
547      * Test method for {@link ensembles.Ensemble#contient(java.lang.Object)}.
548      */
549     @Test
550     public final void testContientENull()
551     {
552         String testName = new String(typeName + ".contient((E)null)");
553         System.out.println(testName);
554         String mot = null;
555 
556         // Contient null sur ensemble vide
557         assertFalse(testName + " ens vide !contient(null) failed",
558                 ensemble.contient(mot));
559 
560         // remplissage ensemble
561         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
562         assertNotNull(testName + " non null instance failed", ensemble);
563         assertEquals(testName + " instance remplie failed",
564                 allSingleElements.length, ensemble.cardinal());
565 
566         // Contient null sur ensemble non vide
567         assertFalse(testName + " ens plein !contient(null) failed",
568                 ensemble.contient((String) null));
569     }
570 
571     /**
572      * Test method for {@link ensembles.Ensemble#contient(java.lang.Object)}.
573      */
574     @Test
```

```
575     public final void testContientE()
576     {
577         String testName = new String(typeName + ".contient(E)");
578         System.out.println(testName);
579         String mot = new String("Bonjour");
580 
581         // Contient mot quelconque sur ensemble vide
582         assertFalse(testName + " ens vide !contient(" + mot + ") failed",
583                 ensemble.contient(mot));
584 
585         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
586         assertNotNull(testName + " non null instance failed", ensemble);
587 
588         // Contient mot quelconque sur ensemble non vide
589         assertFalse(testName + " ens vide contient(" + mot + ") failed",
590                 ensemble.contient(mot));
591 
592         // Contient mots contenus
593         boolean compare = compareElts2Array(testName, ensemble, allSingleElements);
594         assertTrue(testName + " elts compare failed", compare);
595     }
596 
597     /**
598      * Test method for {@link ensembles.Ensemble#contient(ensembles.Ensemble)}.
599      */
600     @Test
601     public final void testContientEnsembleNull()
602     {
603         String testName = new String(typeName + ".contient((Ensemble<E>)null)");
604         System.out.println(testName);
605 
606         // !Contient ensemble null dans ensemble vide
607         assertFalse(testName + "ens vide !contient(null) failed",
608                 ensemble.contient((Ensemble<String>) null));
609 
610         // !Contient ensemble null dans ensemble plein
611         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
612         assertNotNull(testName + " non null instance failed", ensemble);
613         assertEquals(testName + " instance remplie taille failed",
614                 allSingleElements.length, ensemble.cardinal());
615 
616         assertFalse(testName + "ens plein non !contient(null) failed",
617                 ensemble.contient((Ensemble<String>) null));
618     }
619 
620     /**
621      * Test method for {@link ensembles.Ensemble#contient(ensembles.Ensemble)}.
622      */
623     @Test
624     public final void testContientEnsembleOfE()
625     {
626         for (int i = 0; i < typesEnsemble.length; i++)
627         {
628             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
629             String otherTypeName = otherType.getSimpleName();
630 
631             String testName = new String(typeName + ".contient("
632                     + otherTypeName + "<E>)");
633             System.out.println(testName);
634 
635             // sous ensemble vide
636             Ensemble<String> sousEnsemble = constructEnsemble(testName,
637                     typesEnsemble[i], null);
638             assertNotNull(testName + " sousEnsemble non null instance failed",
639                     sousEnsemble);
640 
641             // Contient sous ensemble vide dans ensemble vide
642             assertTrue(testName + " ens vide contient sous ens["
643                     + typesEnsemble[i].getSimpleName() + "] vide failed",
644                     ensemble.contient(sousEnsemble));
645 
646             // remplissage ensemble
647             for (String elt : elements1)
648             {
649                 ensemble.ajout(elt);
650             }
651 
652             // Contient sous ensemble vide dans ensemble non vide
653             assertTrue(testName + " ens plein contient sous ens["
654                     + typesEnsemble[i].getSimpleName() + "] vide failed",
655                     ensemble.contient(sousEnsemble));
656 
```



```
657             // remplissage sous ensemble
658             for (int j = 0; j < (elements1.length / 2); j++)
659             {
660                 sousEnsemble.ajout(elements1[j]);
661             }
662 
663             // Contient sous ensemble non vide ds ens non vide
664             assertTrue(testName + " ens plein contient sous ens["
665                     + typesEnsemble[i].getSimpleName() + "] failed",
666                     ensemble.contient(sousEnsemble));
667 
668             // !Contient sous ensemble non vide non contenu ds ens non vide
669             sousEnsemble.ajout("consectetur");
670             assertFalse(testName + " ens plein !contient sous ens["
671                     + typesEnsemble[i].getSimpleName() + "] failed",
672                     ensemble.contient(sousEnsemble));
673 
674             ensemble.efface();
675         }
676     }
677 
678     /**
679      * Test method for {@link ensembles.Ensemble#efface()}.
680      */
681     @Test
682     public final void testEfface()
683     {
684         String testName = new String(typeName + ".efface()");
685         System.out.println(testName);
686 
687         assertTrue(testName + "ens vide avant effacement failed",
688                 ensemble.estVide());
689 
690         // Effacement ensemble vide
691         ensemble.efface();
692         assertTrue(testName + "ens vide aprÃ¨s effacement failed", ensemble.estVide());
693 
694         // Effacement ensemble non vide
695         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
696         assertNotNull(testName + " non null instance failed", ensemble);
697         assertFalse(testName + "ens non vide aprÃ¨s remplissage failed",
698                 ensemble.estVide());
699         ensemble.efface();
700         assertTrue(testName + "ens vide aprÃ¨s remplissage & effacement failed",
701                 ensemble.estVide());
702     }
703 
704     /**
705      * Test method for {@link ensembles.Ensemble#cardinal()}.
706      */
707     @Test
708     public final void testCardinal()
709     {
710         String testName = new String(typeName + ".cardinal()");
711         System.out.println(testName);
712 
713         assertTrue(testName + " ensemble vide failed", ensemble.estVide());
714         assertEquals(testName + " cardinal 0 sur ensemble vide failed", 0,
715                 ensemble.cardinal());
716 
717         ensemble = constructEnsemble(testName, typeEnsemble, listElements);
718         assertNotNull(testName + " non null instance failed", ensemble);
719 
720         assertFalse(testName + " ensemble non vide failed", ensemble.estVide());
721         assertEquals(testName + " cardinal " + allSingleElements.length
722                 + " sur ensemble rempli failed", allSingleElements.length,
723                 ensemble.cardinal());
724     }
725 
726     /**
727      * Test method for {@link ensembles.Ensemble#union(ensembles.Ensemble)}.
728      */
729     @Test
730     public final void testUnion()
731     {
732         for (int i = 0; i < typesEnsemble.length; i++)
733         {
734             Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
735             String otherTypeName = otherType.getSimpleName();
736 
737             String testName = new String(typeName + ".union(" + otherTypeName
738                     + "<E>)");
```

```
            System.out.println(testName);

            // remplissage ensemble avec singleElements
            for (String elt : elements1)
            {
                ensemble.ajout(elt);
            }

            // remplissage other avec singleElements2
            Ensemble<String> other = constructEnsemble(testName,
                    typesEnsemble[i], null);
            assertNotNull(testName + " other instance non null failed", other);
            for (String elt : elements2)
            {
                other.ajout(elt);
            }

            Ensemble<String> union = ensemble.union(other);

            assertNotNull(testName + " non null union instance failed", union);
            assertFalse(testName + " self union", ensemble ≡ union);
            assertFalse(testName + " self union", other ≡ union);
            assertEquals(testName + " taille failed",
                    allSingleElements.length, union.cardinal());
            boolean compare = compareElts2Array(testName, union,
                    allSingleElements);
            assertTrue(testName + " elts compare failed", compare);
        }
    }

    /**
     * Test method for {@link ensembles.Ensemble#intersection(ensembles.Ensemble)}.
     */
    @Test
    public final void testIntersection()
    {
        for (int i = 0; i < typesEnsemble.length; i++)
        {
            Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
            String otherTypeName = otherType.getSimpleName();

            String testName = new String(typeName + ".intersection("
                    + otherTypeName + "<E>)");
            System.out.println(testName);

            // remplissage ensemble avec singleElements
            for (String elt : elements1)
            {
                ensemble.ajout(elt);
            }

            // remplissage other avec singleElements2
            Ensemble<String> other = constructEnsemble(testName,
                    typesEnsemble[i], null);
            assertNotNull(testName + " other non null instance failed", other);
            for (String elt : elements2)
            {
                other.ajout(elt);
            }

            Ensemble<String> intersection = ensemble.intersection(other);

            assertNotNull(testName + " non null intersection instance failed",
                    intersection);
            assertFalse(testName + " self intersection", ensemble ≡ intersection);
            assertFalse(testName + " self intersection", other ≡ intersection);
            assertEquals(testName + " taille failed",
                    commonSingleElements.length, intersection.cardinal());
            boolean compare = compareElts2Array(testName, intersection,
                    commonSingleElements);
            assertTrue(testName + " elts compare failed", compare);
        }
    }

    /**
     * Test method for {@link ensembles.Ensemble#complement(ensembles.Ensemble)}.
     */
    @Test
    public final void testComplement()
    {
        for (int i = 0; i < typesEnsemble.length; i++)
        {
```



```
            Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
            String otherTypeName = otherType.getSimpleName();

            String testName = new String(typeName + ".complement("
                    + otherTypeName + "<E>)");
            System.out.println(testName);

            // remplissage ensemble avec singleElements
            for (String elt : elements1)
            {
                ensemble.ajout(elt);
            }

            // remplissage other avec singleElements2
            Ensemble<String> other = constructEnsemble(testName,
                    typesEnsemble[i], null);
            assertNotNull(testName + " other non null instance failed", other);
            for (String elt : elements2)
            {
                other.ajout(elt);
            }

            Ensemble<String> complement1 = ensemble.complement(other);

            assertNotNull(testName + " non null complement instance 1 failed",
                    complement1);
            assertFalse(testName + " self complement1", ensemble ≡ complement1);
            assertFalse(testName + " self complement1", other ≡ complement1);
            assertEquals(testName + " taille 1 failed",
                    complementElements1.length, complement1.cardinal());
            boolean compare = compareElts2Array(testName, complement1,
                    complementElements1);
            assertTrue(testName + " elts compare 1 failed", compare);

            Ensemble<String> complement2 = other.complement(ensemble);

            assertNotNull(testName + " non null complement instance 2 failed",
                    complement2);
            assertFalse(testName + " self complement2", ensemble ≡ complement2);
            assertFalse(testName + " self complement2", other ≡ complement2);
            assertEquals(testName + " taille 2 failed",
                    complementElements2.length, complement2.cardinal());
            compare = compareElts2Array(testName, complement2,
                    complementElements2);
            assertTrue(testName + "elts compare 2 failed", compare);
        }
    }

    /**
     * Test method for {@link ensembles.Ensemble#difference(ensembles.Ensemble)}.
     */
    @Test
    public final void testDifference()
    {
        for (int i = 0; i < typesEnsemble.length; i++)
        {
            Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
            String otherTypeName = otherType.getSimpleName();

            String testName = new String(typeName + ".difference("
                    + otherTypeName + "<E>)");
            System.out.println(testName);

            // remplissage ensemble avec singleElements
            for (String elt : elements1)
            {
                ensemble.ajout(elt);
            }

            // remplissage other avec singleElements2
            Ensemble<String> other = constructEnsemble(testName,
                    typesEnsemble[i], null);
            assertNotNull(testName + " other non null instance failed", other);

            for (String elt : elements2)
            {
                other.ajout(elt);
            }

            Ensemble<String> difference = ensemble.difference(other);

            assertNotNull(testName + " difference non null instance failed",
```

```
                    difference);
            assertFalse(testName + " self difference", ensemble ≡ difference);
            assertFalse(testName + " self difference", other ≡ difference);
            assertEquals(testName + " taille failed", diffSingleElements.length,
                    difference.cardinal());
            boolean compare = compareElts2Array(testName, difference,
                    diffSingleElements);
            assertTrue(testName + " elts compare failed", compare);
        }
    }

    /**
     * Test method for {@link ensembles.Ensemble#typeElements()}.
     */
    @Test
    public final void testTypeElements()
    {
        String testName = new String(typeName + ".typeElements()");
        System.out.println(testName);

        assertNotNull(testName + " non null instance failed", ensemble);

        // type elt sur ensemble vide == null
        assertEquals(testName + " sur ens vide failed", null,
                ensemble.typeElements());

        // type elt sur ensemble non vide == String
        ensemble = constructEnsemble(testName, typeEnsemble, listElements);
        assertNotNull(testName + " non null instance failed", ensemble);
        assertEquals(testName + " sur ens non vide failed", String.class,
                ensemble.typeElements());
    }

    /**
     * Test method for {@link ensembles.Ensemble#equals(java.lang.Object)}.
     */
    @Test
    public final void testEquals()
    {
        String testName = new String(typeName + ".equals(Object)");
        System.out.println(testName);

        // Equals sur null
        assertFalse(testName + " sur null failed", ensemble.equals(null));

        // Equals sur this
        assertTrue(testName + " sur this failed", ensemble.equals(ensemble));

        // Equals sur autre objet
        assertFalse(testName + " sur Object failed",
                ensemble.equals(new Object()));

        // remplissage ensemble
        for (String elt : allSingleElementsSorted)
        {
            ensemble.ajout(elt);
        }

        String[] allsingleElementsShuffle = shuffleElements(allSingleElements);

        for (int i = 0; i < typesEnsemble.length; i++)
        {
            Class<? extends Ensemble<String>> otherType = typesEnsemble[i];
            String otherTypeName = otherType.getSimpleName();

            Ensemble<String> other = constructEnsemble(testName,
                    typesEnsemble[i], null);

            // Equals sur Ensemble mÃªme contenu mÃªme ordre
            assertNotNull(testName + " other non null instance failed", other);
            for(String elt : allSingleElementsSorted)
            {
                other.ajout(elt);
            }
            assertEquals(testName + " ens identique, ordre identique["
                    + otherTypeName + "] failed", ensemble, other);

            // Equals sur Ensemble mÃªme contenu ordre diffÃ©rent
            other.efface();
            for(String elt : allsingleElementsShuffle)
            {
                other.ajout(elt);
```



```
            }

            // ensemble est toujours sorted car construit avec
            // allSingleElementsSorted
            if ((ensemble instanceof EnsembleTri<?>) ∧
               ¬(other instanceof EnsembleTri<?>))
            {
                assertFalse(testName + " ens identique, ordre diffÃ©rent["
                        + otherTypeName + "] failed", ensemble.equals(other));
            }
            else
            {
                assertEquals(testName + " ens identique, ordre diffÃ©rent["
                        + otherTypeName + "] failed", ensemble, other);
            }

            // Equals sur Ensemble contenu diffÃ©rent
            other.ajout("bonjour");
            assertFalse(testName + " ens diffÃ©rent failed",
                    ensemble.equals(other));
        }
    }

    /**
     * Test method for {@link ensembles.Ensemble#hashCode()}.
     */
    @Test
    public final void testHashCode()
    {
        String testName = new String(typeName + ".hashCode()");
        System.out.println(testName);
        int hash;
        boolean trie = ensemble instanceof EnsembleTri<?>;
        if (trie)
        {
            hash = 1;
        }
        else
        {
            hash = 0;
        }

        // hash code ensemble vide ==
        //  0 pour les Ensemble
        //  1 pour les EnsembleTri
        assertEquals(testName + " hashcode ens vide failed", hash,
                ensemble.hashCode());

        // hash code ensemble non vide ==
        //  somme des hashcode des elts pour les Ensemble
        //  comme les collections pour les EnsembleTri
        for (String elt : allSingleElements)
        {
            ensemble.ajout(elt);
        }
        if (trie)
        {
            final int prime = 31;
            for (String elt : allSingleElementsSorted)
            {
                hash = (prime * hash) + (elt ≡ null ? 0 : elt.hashCode());
            }
        }
        else
        {
            for (String elt : allSingleElements)
            {
                hash += elt.hashCode();
            }
        }

        assertEquals(testName + " hashcode ens non vide failed", hash,
                ensemble.hashCode());
    }

    /**
     * Test method for {@link ensembles.Ensemble#toString()}.
     */
    @Test
    public final void testToString()
    {
        String testName = new String(typeName + ".toString()");
```

```
1067        System.out.println(testName);
1068
1069        ensemble = constructEnsemble(testName, typeEnsemble, listElements);
1070        assertNotNull(testName + " non null instance failed", ensemble);
1071
1072        StringBuilder sb = new StringBuilder();
1073        sb.append("[");
1074        Iterator<String> it = ensemble.iterator();
1075        if (it ≠ null)
1076        {
1077            for (; it.hasNext();)
1078            {
1079                sb.append(it.next().toString());
1080                if (it.hasNext())
1081                {
1082                    sb.append(", ");
1083                }
1084            }
1085            sb.append("]");
1086
1087            String expected = sb.toString();
1088
1089            assertEquals(testName, expected, ensemble.toString());
1090        }
1091        else
1092        {
1093            fail(testName + " null iterator");
1094        }
1095    }
1096
1097    /**
1098     * Test method for {@link ensembles.Ensemble#iterator()}.
1099     */
1100    @Test
1101    public final void testIterator()
1102    {
1103        String testName = new String(typeName + ".iterator()");
1104        System.out.println(testName);
1105
1106        Iterator<String> it = null;
1107
1108        // iterator existe
1109        it = ensemble.iterator();
1110        assertNotNull(testName +  " non null instance failed", it);
1111
1112        // iterator sur ens vide n'a pas d'elts Ã  itÃ©rer
1113        assertFalse(testName + " !hasNext() sur ens vide failed", it.hasNext());
1114
1115        // remplissage
1116        for (String elt : allSingleElements)
1117        {
1118            ensemble.ajout(elt);
1119        }
1120
1121        it = ensemble.iterator();
1122
1123        // iterator sur ens rempli
1124        assertTrue(testName + " hashNext() sur ens rempli failed", it.hasNext());
1125
1126        String[] array;
1127        if (ensemble instanceof EnsembleTri<?>)
1128        {
1129            array = allSingleElementsSorted;
1130        }
1131        else
1132        {
1133            array = allSingleElements;
1134        }
1135
1136        // comparaison des elts
1137        for(int i = 0; (i < array.length) ∧ it.hasNext(); i++)
1138        {
1139            assertEquals(testName + "check elt:" + array[i] + " failed",
1140                    array[i], it.next());
1141        }
1142
1143        // plus l'elts Ã  itÃ©rer
1144        assertFalse(testName + " !hasNext() fin comparaison failed",
1145                it.hasNext());
1146
1147        // retrait des elts avec l'itÃ©rateur
1148        it = ensemble.iterator();
```



```
1149        for (int i = 0; (i < array.length) ∧ it.hasNext(); i++)
1150        {
1151            it.next();
1152            it.remove();
1153            assertFalse(testName + " retrait elt:" + array[i] + " failed",
1154                    ensemble.contient(array[i]));
1155        }
1156
1157        // plus l'elts Ã  itÃ©rer
1158        assertFalse(testName + " !hasNext() fin retrait failed", it.hasNext());
1159        assertTrue(testName + " ens vide aprÃ¨s retraits failed",
1160                ensemble.estVide());
1161    }
1162 }
```

```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.HashMap;
import java.util.Iterator;
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.EnsembleTri;
import ensembles.EnsembleTriFactory;
import ensembles.EnsembleTriTableau;

/**
 * Classe de test complÃ©mentaire pour tous les types d'ensembles triÃ©s :
 * {@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 EnsembleTriTest
{
    /**
     * l'ensemble Ã  tester
     */
    private EnsembleTri<String> ensemble;

    /**
     * Le type d'ensemble Ã  tester.
     */
    private Class<? extends EnsembleTri<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 EnsembleTri<String>>[] typesEnsemble =
    (Class<? extends EnsembleTri<String>>[]) new Class<?>[]
    {
        /*
         * TODO Commenter / dÃ©commenter les lignes ci-dessous en focntion
         * de votre avancement (Attention la derniÃ¨re ligne non commentÃ©e
         * ne doit pas avoir de virgule)
         */
//      EnsembleTriVector.class,
//      EnsembleTriVector2.class,
        EnsembleTriTableau.class
//      EnsembleTriTableau2.class,
//      EnsembleTriListe.class,
//      EnsembleTriListe2.class
    };

    /**
     * Elements pour remplir l'ensemble
     */
    private static final String[] elements = new String[] {
        "Lorem",        // 0
        "ipsum",        // 6
        "sit",          // 7
        "dolor",        // 4
```



```java
        "amet",         // 2
        "dolor",        // 4
        "amet",         // 2
        "consectetur",  // 3
        "adipisicing",  // 1
        "elit"          // 5
    };

    /**
     * Rang d'insertion des Ã©lÃ©ments successifs
     */
    private static final int[] insertionRank = new int[] {
        0,  // Lorem
        1,  // ipsum
        2,  // sit
        1,  // dolor
        1,  // amet
        2,  // dolor
        1,  // amet
        2,  // consectetur
        1,  // adipisicing
        5,  // elit
    };
    /**
     * Elements triÃ©s pour contrÃ´ler le remplissage de l'ensemble
     */
    private static final String[] singleSortedElements = new String[] {
        "Lorem",        // 0
        "adipisicing",  // 1
        "amet",         // 2
        "consectetur",  // 3
        "dolor",        // 4
        "elit",         // 5
        "ipsum",        // 6
        "sit"           // 7
    };

    /**
     * Elements triÃ©s pour contrÃ´ler le remplissage de l'ensemble
     */
    private static final String[][] insertSortedElements = new String[][] {
        {"Lorem"},
        {"Lorem", "ipsum"},
        {"Lorem", "ipsum", "sit"},
        {"Lorem", "dolor", "ipsum", "sit"},
        {"Lorem", "amet", "dolor", "ipsum", "sit"},
        {"Lorem", "amet", "dolor", "ipsum", "sit"},
        {"Lorem", "amet", "dolor", "ipsum", "sit"},
        {"Lorem", "amet", "consectetur", "dolor", "ipsum", "sit"},
        {"Lorem", "adipisicing", "amet", "consectetur", "dolor", "ipsum", "sit"},
        singleSortedElements
    };

    /**
     * Collection pour contenir les Ã©lements de remplissage
     */
    private ArrayList<String> listElements;

    /**
     * Construit une instance de EnsembleTri<String> en fonction d'un type
     * d'ensemble Ã  crÃ©er et Ã©ventuellement d'un contenu l'ensemble Ã  mettre en
     * place
     *
     * @param testName le message Ã  rÃ©pÃ©ter dans les assertions en fonction du
     *            test dans lequel est employÃ© cette mÃ©thode
     * @param type le type d'ensemble Ã  crÃ©er
     * @param content le contenu Ã  mettre en place dans le nouvel ensemble, ou
     * bien null si aucun contenu n'est requis.
     * @return un nouvel ensemble du type demandÃ© evt rempli avec le contenu
     * fournit s'il est non null.
     */
    private static EnsembleTri<String>
    constructEnsemble(String testName,
                      Class<? extends EnsembleTri<String>> type,
                      Iterable<String> content)
    {
        EnsembleTri<String> ensemble = null;

        try
        {
            ensemble = EnsembleTriFactory.<String>getEnsemble(type, content);
        }
```

```
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 paramÃªtrÃ© par le type d'ensemble Ã  tester.
290      * LancÃ© pour chaque test
291      * @param typeEnsemble le type d'ensemble Ã  gÃ©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    }
```