```
# Executables
OSTYPE = $(shell uname -s)
JAVAC = javac
JAVA = java
A2PS = a2ps-utf8
GHOSTVIEW = gv
DOCP = javadoc
ARCH = zip
PS2PDF = ps2pdf -dPDFX=true -sPAPERSIZE=a4
DATE = $(shell date +%Y-%m-%d)
# Options de compilation
#CFLAGS = -verbose
CFLAGS =
CLASSPATH=.

JAVAOPTIONS = --verbose

PROJECT=Figures
# nom du fichier d'impression
OUTPUT = $(PROJECT)
# nom du répertoire ou se situera la documentation
DOC = doc
# lien vers la doc en ligne du JDK
WEBLINK = "http://docs.oracle.com/javase/8/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
# Répertoire source
SRC = src
# Répertoire bin
BIN = bin
# Répertoire Listings
LISTDIR = listings
# Répertoire Archives
ARCHDIR = archives
# Répertoire Figures
FIGDIR = graphics
# noms des fichiers sources
MAIN = TestListe TestFigures RunAllTests
SOURCES = \
$(foreach name, $(MAIN), $(SRC)/$(name).java) \
$(SRC)/listes/package-info.java \
$(SRC)/listes/IListe.java \
$(SRC)/listes/Liste.java \
$(SRC)/listes/CollectionListe.java \
$(SRC)/points/package-info.java \
$(SRC)/points/Point2D.java \
$(SRC)/points/Vecteur2D.java \
$(SRC)/figures/package-info.java \
$(SRC)/figures/Figure.java \
$(SRC)/figures/AbstractFigure.java \
$(SRC)/figures/Cercle.java \
$(SRC)/figures/Rectangle.java \
$(SRC)/figures/Triangle.java \
$(SRC)/figures/Polygone.java \
$(SRC)/figures/Groupe.java \
$(SRC)/tests/package-info.java \
$(SRC)/tests/AllTests.java \
$(SRC)/tests/ListeTest.java \
$(SRC)/tests/CollectionListeTest.java \
$(SRC)/tests/Point2DTest.java \
$(SRC)/tests/FigureTest.java

OTHER =

.PHONY : doc ps

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

#règle de compilation générique
$(BIN)/%.class : $(SRC)/%.java
	$(JAVAC) -sourcepath $(SRC) -classpath $(BIN):$(CLASSPATH) -d $(BIN) $(CFLAGS) $<

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

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

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

```



```
# générer 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

# générer le listing lisible pour Gérard
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)~

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

# générer une archive de sauvegarde
$(ARCHDIR) :
	mkdir $(ARCHDIR)

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

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

```java
import java.util.ArrayList;

import listes.CollectionListe;
import listes.Liste;

/**
 * Classe de test de la Liste et de la CollectionListe
 *
 * @author davidroussel
 */
public class TestListe
{

    /**
     * Programme principal de test des {@link Liste} et {@link CollectionListe}
     *
     * @param args arguments (non utilisés ici)
     */
    public static void main(String[] args)
    {
        String[] mots = { "mot1", "mot2", "mot3", "mot4", "mot5", "mot6",
                "mot7" };

        // ------------------------------------------------------------------
        // Liste<String>
        // ------------------------------------------------------------------
        Liste<String> liste1 = new Liste<String>();
        int count = 1;
        for (String mot : mots)
        {
            System.out.print("Ajout de " + mot);
            if ((count%2) == 0)
            {
                liste1.add(mot);
                System.out.println(" à la fin");
            }
            else
            {
                liste1.insert(mot);
                System.out.println(" au début");
            }
            count++;
        }

        System.out.println("Liste = " + liste1);

        Liste<String> liste2 = new Liste<String>(liste1);

        System.out.print("Comparaison de " + liste1 + " et " + liste2 + " :");
        System.out.println(liste1.equals(liste2) ? "Ok" : "Ko");

        System.out.print("Comparaison de " + liste1 + " et " + mots + " :");
        System.out.println(liste1.equals(mots) ? "Ok" : "Ko");


        for (int i=0; i < mots.length; i++)
        {
            liste1.remove(mots[i]);
            System.out.println("Liste – " + mots[i] + " = " + liste1);
        }

        liste2.clear();
        System.out.println("Liste2 après effacement : " + liste2);

        liste1.insert(mots[0], 0);
        liste1.insert(mots[6], 1);
        liste1.insert(mots[0], -1);
        liste1.insert(mots[0], 3);
        liste1.insert(mots[1], 1);
        liste1.insert(mots[4], 2);
        liste1.insert(mots[2], 2);
        liste1.insert(mots[5], 4);
        System.out.println("Liste1 après insertion indexée : " + liste1);

        // ------------------------------------------------------------------
        // CollectionListe<String>
        // ------------------------------------------------------------------

        CollectionListe<String> colListe1 = new CollectionListe<String>();
        ArrayList<String> vector1 = new ArrayList<String>();
        for (String mot : mots)
        {
            colListe1.add(mot);
            vector1.add(mot);
        }

        System.out.println("Collection Liste   : " + colListe1 + ", hash = "
                + colListe1.hashCode());
        System.out.println("Collection standard : " + vector1 + ", hash = "
                + vector1.hashCode());
```



```java

        System.out.print("La Collection Liste est ");
        if (colListe1.equals(vector1))
        {
            System.out.print("égale au");
        }
        else
        {
            System.out.print("différente du");
        }
        System.out.println(" ArrayList en terme de contenu");

        vector1.remove("mot7");

        System.out.println("Collection Liste   : " + colListe1 + ", hash = "
                + colListe1.hashCode());
        System.out.println("Collection standard : " + vector1 + ", hash = "
                + vector1.hashCode());

        System.out.print("La Collection Liste est ");
        if (colListe1.equals(vector1))
        {
            System.out.print("égale au");
        }
        else
        {
            System.out.print("différente du");
        }
        System.out.println(" ArrayList en terme de contenu");

        CollectionListe<String> coListe2 = new CollectionListe<String>(
                colListe1);

        System.out.println("Collection Liste 1 : " + colListe1 + ", hash = "
                + colListe1.hashCode());
        System.out.println("Collection Liste 2 : " + coListe2 + ", hash = "
                + coListe2.hashCode());

        System.out.print("La Collection Liste est ");
        if (colListe1.equals(coListe2))
        {
            System.out.print("égale à");
        }
        else
        {
            System.out.print("différente de");
        }
        System.out.println(" l'autre Collection Liste en terme de contenu");
    }

}
```

```java
import java.util.ArrayList;
import java.util.Collection;

import listes.CollectionListe;
import points.Point2D;
import figures.AbstractFigure;
import figures.Cercle;
import figures.Figure;
import figures.Polygone;
import figures.Rectangle;
import figures.Triangle;

/**
 * Class de test des Figures
 * @author davidroussel
 */
class TestFigures
{
    /**
     * Programme de test des figures
     * @param args arguments (non utilisés)
     */
    public static void main (String args[])
    {
        Cercle cer, cer2;
        Rectangle rec, rec2;
        Triangle tri, tri2;

        // création d'un cercle
        Point2D centre = new Point2D(0, 0);
        cer = new Cercle(centre, 2);
        cer2 = new Cercle(cer);
        System.out.println(cer + " == " + cer2 + " ? :" + cer.equals(cer2));

        // création d'un carré centré en 2.5, 1.5
        Point2D pmin = new Point2D(2,1);
        Point2D pmax = new Point2D(3,2);
        rec = new Rectangle(pmin,pmax);
        rec2 = new Rectangle(rec);
        System.out.println(rec + " == " + rec2 + " ? :" + rec.equals(rec2));

        // création d'un triangle
        tri = new Triangle();
        tri2 = new Triangle(tri);
        System.out.println(tri + " == " + tri2 + " ? :" + tri.equals(tri2));

        // création d'un polygone
        Point2D p0 = new Point2D(4,1);
        Point2D p1 = new Point2D(4,1);
        Point2D p2 = new Point2D(5,3);
        Point2D p3 = new Point2D(4,5);
        Point2D p4 = new Point2D(2,5);
        Polygone poly = new Polygone(p0,p1);
        poly.ajouter(p2);
        poly.ajouter(p3);
        poly.ajouter(p4);
        Polygone poly2 = new Polygone(poly);
        System.out.println(poly + " == " + poly2 + " :"
                + (poly.equals(poly2) ? "Ok" : "Ko"));

        ArrayList<Point2D> vp = new ArrayList<Point2D>();
        vp.add(p0);
        vp.add(p1);
        vp.add(p2);
        vp.add(p3);
        vp.add(p4);
        Polygone poly3 = new Polygone(vp);
        System.out.println(poly + " == " + poly3 + " :"
                + (poly.equals(poly3) ? "Ok" : "Ko"));

        // création d'une ligne
        Point2D pl0 = new Point2D(0,0);
        Point2D pl1 = new Point2D(1,0);
        Polygone ligne = new Polygone(pl0,pl1);
        Polygone ligne2 = new Polygone(ligne);
        System.out.println(ligne + " == " + ligne2 + " ? :" + ligne.equals(ligne2));


        // test des différentes méthodes communes au figures
        Collection<Figure> figures= new CollectionListe<Figure>();
        figures.add(cer);
        figures.add(rec);
        figures.add(tri);
        figures.add(poly);
        System.out.println("Ma " + figures);

        // affichage
        for (Figure f : figures)
        {
            System.out.println(f);
        }
```



```java
        }

        // déplacement
        for (Figure f : figures)
        {
            f.deplace(1,1);
        }

        // nouvel affichage après déplacement
        for (Figure f : figures)
        {
            System.out.println(f);
        }

        // test de contenu
        Point2D pcont = new Point2D(2,2);
        pcont.deplace(-0.5, -0.75);
        System.out.println("Test de contenance du point " + pcont);
        System.out.println("Le point " + pcont + " est : ");
        for (Figure f : figures)
        {
            afficheContenance(f, pcont);
        }

        Point2D pcont2 = new Point2D(3,3);
        System.out.println("Test de contenance du point " + pcont2);
        System.out.println("Le point " + pcont2 + " est : ");
        for (Figure f : figures)
        {
            afficheContenance(f, pcont2);
        }

        Point2D pcont3 = new Point2D(0.5, 0);
        System.out.println("Test de contenance du point " + pcont3);
        System.out.println("Le point " + pcont3 + " est : ");
        for (Figure f : figures)
        {
            afficheContenance(f, pcont3);
        }

        // distance aux centres
        Collection<Figure> figures2 = new CollectionListe<Figure>(figures);
        for (Figure f1 : figures)
        {
            for (Figure f2 : figures2)
            {
                afficheDistanceCentres(f1,f2);
            }
        }

        // aires des figures
        figures.add(ligne);
        for (Figure f : figures)
        {
            afficheAire(f);
        }

        // ajout d'une deuxième occurrence de polygone dans la collection
        figures.add(poly);
        System.out.println("Ma " + figures + " avant retrait de " + poly);

        // retrait de toutes les occurences de poly de la collection
        int count = 0;
        while (figures.contains(poly))
        {
            if (figures.remove(poly))
            {
                count++;
            }
        }

        // affichage de la collection après retrait des poly
        System.out.print("Ma " + figures + "après retrait : ");
        System.out.println(" " + count + " occurrences supprimées");

    }

    public static void afficheContenance(Figure f, Point2D p)
    {
        if (f.contient(p))
        {
            System.out.println("     dans le " + f.getNom());
        }
        else
        {
            System.out.println("     en dehors du " + f.getNom());
        }
    }

    public static void afficheDistanceCentres(Figure f1, Figure f2)
```

```
181     {
182         System.out.println("Distance " + f1.getNom() +
183                            " -> " + f2.getNom() +
184                            ":" + Figure.distanceToCentre(f1,f2));
185     }
186
187     public static void afficheAire(Figure f)
188     {
189         System.out.println("Aire de " + f.getNom() + ":" + f.aire());
190     }
191
192 }
```



```
1  import org.junit.runner.JUnitCore;
2  import org.junit.runner.Result;
3  import org.junit.runner.notification.Failure;
4
5  import tests.AllTests;
6
7  /**
8   * Exécution de tous les tests du TP1
9   *
10  * @author davidroussel
11  */
12 public class RunAllTests
13 {
14
15     /**
16      * Programme principal de lancement des tests
17      * @param args non utilisés
18      */
19     public static void main(String[] args)
20     {
21         System.out.println("Tests du TP1");
22
23         Result result = JUnitCore.runClasses(AllTests.class);
24
25         int failureCount = result.getFailureCount();
26
27         if (failureCount == 0)
28         {
29             System.out.println("Every thing went fine");
30         }
31         else
32         {
33             for (Failure failure : result.getFailures())
34             {
35                 System.err.println("Failure : " + failure.toString());
36             }
37         }
38     }
39 }
```

```java
/**
 * Package contenant l'ensemble des tests JUnit4
 */
package tests;
```



```java
package listes;

import java.util.Iterator;

/**
 * Interface d'une liste générique d'éléments.
 *
 * @note On considèrera que la liste ne peut pas contenir d'elt null
 * @author David Roussel
 * @param <E> le type des éléments de la liste.
 */
public interface IListe<E> extends Iterable<E>
{
    /**
     * Ajout d'un élément en fin de liste
     *
     * @param elt l'élément à ajouter en fin de liste
     * @throws NullPointerException si l'on tente d'ajouter un élément null
     */
    public abstract void add(E elt) throws NullPointerException;

    /**
     * Insertion d'un élément en tête de liste
     *
     * @param elt l'élément à ajouter en tête de liste
     * @throws NullPointerException si l'on tente d'insérer un élement null
     */
    public abstract void insert(E elt) throws NullPointerException;

    /**
     * Insertion d'un élément à la (index+1)ième place
     *
     * @param elt l'élément à insérer
     * @param index l'index de l'élément à insérer
     * @return true si l'élément a pu être inséré à l'index voulu, false sinon
     *         ou si l'élément à insérer était null
     */
    public abstract boolean insert(E elt, int index);

    /**
     * Suppression de la première occurrence de l'élémént e
     *
     * @param elt l'élement à rechercher et à supprimer.
     * @return true si l'élément a été trouvé et supprimé de la liste
     * @note doit fonctionner même si e est null
     */
    public default boolean remove(E elt)
    {
        // TODO Remplacer par l'implémentation ...
        return false;
    }

    /**
     * Suppression de toutes les instances de e dans la liste
     *
     * @param elt l'élément à supprimer
     * @return true si au moins un élément a été supprimé
     * @note doit fonctionner même si e est null
     */
    public default boolean removeAll(E elt)
    {
        boolean result = false;
        // TODO Compléter par l'implémentation ...
        return result;
    }

    /**
     * Nombre d'éléments dans la liste
     *
     * @return le nombre d'éléments actuellement dans la liste
     */
    public default int size()
    {
        int count = 0;
        // TODO Compléter par l'implémentation ...
        return count;
    }

    /**
     * Effacement de la liste;
     */
    public default void clear()
    {
        // TODO Remplacer par l'implémentation ...
    }

    /**
     * Test de liste vide
     *
     * @return true si la liste est vide, false sinon
```

```
91      */
92     public default boolean empty()
93     {
94         // TODO remplacer par l'implémetation
95         return true;
96     }
97
98     /**
99      * Test d'égalité au sens du contenu de la liste
100     *
101     * @param o la liste dont on doit tester le contenu
102     * @return true si o est une liste, que tous les maillons des deux listes
103     *         sont identiques (au sens du equals de chacun des maillons), dans
104     *         le même ordre, et que les deux listes ont la même longueur. false
105     *         sinon
106     * @note On serait tenté d'en faire une "default method" dans la mesure où
107     *       l'on peut n'utiliser que l'itérateur pour parcourir les éléments de
108     *       la liste MAIS les méthodes par défaut n'ont pas le droit de
109     *       surcharger les méthodes de la superclasse Object.
110     */
111    @Override
112    public abstract boolean equals(Object o);
113
114    /**
115     * HashCode d'une liste
116     *
117     * @return le hashCode de la liste
118     * @note On serait tenté d'en faire une "default method" dans la mesure où
119     *       l'on peut n'utiliser que l'itérateur pour parcourir les éléments de
120     *       la liste MAIS les méthodes par défaut n'ont pas le droit de
121     *       surcharger les méthodes de la superclasse Object.
122     */
123    @Override
124    public abstract int hashCode();
125
126    /**
127     * Représentation de la chaine sous forme de chaine de caractère.
128     *
129     * @return une chaine de caractère représentant la liste chainée
130     * @note On serait tenté d'en faire une "default method" dans la mesure où
131     *       l'on peut n'utiliser que l'itérateur pour parcourir les éléments de
132     *       la liste MAIS les méthodes par défaut n'ont pas le droit de
133     *       surcharger les méthodes de la superclasse Object.
134     */
135    @Override
136    public abstract String toString();
137
138    /**
139     * Obtention d'un itérateur pour parcourir la liste : <code>
140     * Liste<Type> l = new Liste<Type>();
141     * ...
142     * for (Iterator<Type> it = l.iterator(); it.hasNext(); )
143     * {
144     *      ... it.next() ...
145     * }
146     * ou bien
147     * for (Type elt : l)
148     * {
149     *      ... elt ...
150     * }
151     * </code>
152     *
153     * @return un nouvel itérateur sur la liste
154     * @see {@link Iterable#iterator()}
155     */
156    @Override
157    public abstract Iterator<E> iterator();
158 }
```



```
1  /**
2   * Package contenant l'ensemble des tests JUnit4
3   */
4  package tests;
```

```java
package points;


/**
 * Classe définissant un point du plan 2D
 *
 * @author David Roussel
 */
public class Point2D
{
    // attributs d'instance ------------------------------------------------
    /**
     * l'abcisse du point
     */
    protected double x;
    /**
     * l'ordonnée du point
     */
    protected double y;

    // attributs de classe -------------------------------------------------
    /**
     * Compteur d'instances : le nombre de points actuellement instanciés
     */
    protected static int nbPoints = 0;

    /**
     * Constante servant à comparer deux points entre eux (à {@value} près). On
     * comparera alors la distance entre deux points.
     *
     * @see #distance(Point2D)
     * @see #distance(Point2D, Point2D)
     */
    protected static final double epsilon = 1e-6;

    /*
     * Constructeurs
     */
    /**
     * Constructeur par défaut. Initialise un point à l'origine du repère [0.0,
     * 0.0]
     */
    public Point2D()
    {
        // utilisation du constructeur valué dans le constructeur par défaut
        // Attention si un constructeur est utilisé dans un autre constructeur
        // il doit être la PREMIERE instruction de ce constructeur
        // (Obligatoirement)
        this(0.0, 0.0);
    }

    /**
     * Constructeur valué
     *
     * @param x l'abcisse du point à créer
     * @param y l'ordonnée du point à créer
     */
    public Point2D(double x, double y)
    {
        this.x = x;
        this.y = y;
        nbPoints++;
    }

    /**
     * Constructeur de copie
     *
     * @param p le point dont il faut copier les coordonnées Il s'agit ici d'une
     *            copie profonde de manière à créer une nouvelle instance
     *            possédant les même caractéristiques que celle dont on copie
     *            les coordonnées.
     */
    public Point2D(Point2D p)
    {
        // utilisation du constructeur valué dans le constructeur par défaut
        this(p.x, p.y);
    }

    /**
     * Nettoyeur avant destruction Permet de décrémenter le compteur d'instances
     */
    @Override
    protected void finalize()
    {
        nbPoints--;
    }

    /*
     * Accesseurs
     */
```



```java
    /**
     * Accesseur en lecture de l'abcisse
     *
     * @return l'abcisse du point.
     */
    public double getX()
    {
        return x;
    }

    /**
     * Accesseur en lecture de l'ordonnée
     *
     * @return l'ordonnée du point.
     */
    public double getY()
    {
        return y;
    }

    /**
     * Accesseur en écriture de l'abcisse
     *
     * @param val valeur à donner à l'abcisse
     */
    public void setX(double val)
    {
        x = val;
    }

    /**
     * Accesseur en écriture de l'ordonnée
     *
     * @param val valeur à donner à l'ordonnée
     */
    public void setY(double val)
    {
        y = val;
    }

    /**
     * Accesseur en lecture d'epsilon
     *
     * @return la valeur d'epsilon choisie pour comparer deux grandeurs à
     *         epsilon près.
     * @note Dans la mesure où epsilon est une constante qui ne peut pas changer
     * de valeur, il est tout à fait concevable de la rendre publique ce qui
     * éviterait cet accesseur
     */
    public static double getEpsilon()
    {
        return epsilon;
    }

    /**
     * Accesseur en lecture du nombre de points actuellement instanciés
     *
     * @return le nombre de points actuellement instanciés
     */
    public static int getNbPoints()
    {
        return nbPoints;
    }

    /*
     * Affichage contenu
     */
    // toString est une méthode classique en Java, elle est présente
    // dans les objets de type Object, on pourra donc ainsi l'utiliser
    // dans une éventuelle Liste de points.
    /**
     * Méthode nécessaire pour l'affichage qui permet de placer un point dans un
     * {@link java.io.PrintStream#println()} comme {@link System#out}.
     *
     * @return une chaîne de caractères représentant un point.
     */
    @Override
    public String toString()
    {
        return new String("x=" + x + "y=" + y);
    }

    /**
     * Opérations sur un point
     *
     * @param dx le déplacement en x
     * @param dy le déplacement en y
     * @return renvoie la référence vers l'instance courante (this) de manière à
     *         pouvoir enchainer les traitements du style :
     *         unObjet.uneMéthode(monPoint.deplace(dx,dy))
```

```
181      */
182     public Point2D deplace(double dx, double dy)
183     {
184         x += dx;
185         y += dy;
186         return this;
187     }
188
189     /*
190      * Méthodes de classe : opérations sur les points
191      */
192     /**
193      * Calcul de l'écart en abcsisse entre deux points. Cet écart ne concerne
194      * pas plus le premier que le second point c'est pourquoi on en fait une
195      * méthode de classe.
196      *
197      * @param p1 le premier point
198      * @param p2 le second point
199      * @return l'écart en x entre les deux points
200      */
201     protected static double dx(Point2D p1, Point2D p2)
202     {
203         return (p2.x - p1.x);
204     }
205
206     /**
207      * Calcul de l'écart en ordonnée entre deux points. Cet écart ne concerne
208      * pas plus le premier que le second point c'est pourquoi on en fait une
209      * méthode de classe.
210      *
211      * @param p1 le premier point
212      * @param p2 le second point
213      * @return l'écart en y entre les deux points
214      */
215     protected static double dy(Point2D p1, Point2D p2)
216     {
217         return (p2.y - p1.y);
218     }
219
220     /**
221      * Calcul de la distance 2D entre deux points. Cette distance ne concerne
222      * pas plus un point que l'autre c'est pourquoi on en fait une méthode de
223      * classe. Cette méthode utilise les méthodes {@link #dx(Point2D, Point2D)}
224      * et {@link #dy(Point2D, Point2D)} pour calculer la distance entre les
225      * points.
226      *
227      * @param p1 le premier point
228      * @param p2 le seconde point
229      * @return la distance entre les points p1 et p2
230      * @see #dx(Point2D, Point2D)
231      * @see #dy(Point2D, Point2D)
232      */
233     public static double distance(Point2D p1, Point2D p2)
234     {
235         // on remarquera que là aussi on
236         // utilise des méthodes statiques
237         // de l'objet Math : sqrt ou hypot
238
239         double dx = dx(p1, p2);
240         double dy = dy(p1, p2);
241
242         return (Math.hypot(dx, dy) - 1.0);
243     }
244
245     /**
246      * Calcul de distance 2D par rapport au point courant
247      *
248      * @param p l'autre point dont on veut calculer la distance
249      * @return la distance entre le point courant et le point p
250      * @see #distance(Point2D, Point2D)
251      */
252     public double distance(Point2D p)
253     {
254         return distance(this, p);
255     }
256
257     /**
258      * Test d'égalité entre deux points 2D. Deux points sont considérés comme
259      * identiques si leur distance est inférieure à {@link #epsilon}.
260      * Cette méthode n'est utilisée que dans {@link #equals(Object)} donc elle
261      * n'est pas publique.
262      *
263      * @param p le point dont on veut tester l'égalité par rapport au point
264      *           courant
265      * @return true si les points sont plus proches que {@link #epsilon}, false
266      *         sinon.
267      */
268     protected boolean equals(Point2D p)
269     {
270         // version distance
```



```
271         return (distance(p) < epsilon);
272     }
273
274     /**
275      * Test d'égalité générique (hérité de la classe Object)
276      *
277      * @param o le point à tester (si c'est bien un point)
278      * @return true si les points sont plus proches que {@link #epsilon}, false
279      *         sinon ou bien si l'argument n'est pas un point. Il est important
280      *         d'implémenter cette version de la comparaison car lorsque de tels
281      *         points seront contenus dans des conteneurs génériques comme des
282      *         {@link java.util.Vector} ou des {@link listes.Liste} seule
283      *         cette comparaison pourra être utilisée.
284      * @note il est possible que l'on ne puisse pas faire ceci dans le premier
285      *       TD car on aura pas encore vu l'introspection
286      */
287     @Override
288     public boolean equals(Object o)
289     {
290         if (o == null)
291         {
292             return false;
293         }
294         if (o == this)
295         {
296             return true;
297         }
298         //  comparaison laxiste (les points 2D et leurs héritiers)
299         // if (this.getClass().isInstance(o))
300         //  comparaison stricte (uniquement les Points 2D)
301         if (this.getClass().equals(o.getClass()))
302         {
303             return equals((Point2D) o);
304         }
305         else
306         {
307             return false;
308         }
309     }
310 }
```

```java
package points;

/**
 * Classe définissant un vecteur du plan
 * @author davidroussel
 */
public class Vecteur2D extends Point2D
{

    /**
     * Constructeur par défaut d'un vecteur 2D : construit un vecteur nul
     */
    public Vecteur2D()
    {
        super();
    }

    /**
     * Constructeur valué d'un vecteur 2D à partir d'un point2D : construit
     * le vecteur reliant l'origine à ce point
     * @param pt le point fournissant les coordonnées du vecteur
     */
    public Vecteur2D(Point2D pt)
    {
        super(pt);
    }

    /**
     * Constructeur valué d'un vecteur 2D à partir de coordonnées brutes
     * @param x l'ordonnée du vecteur
     * @param y l'abcisse du vecteur
     */
    public Vecteur2D(double x, double y)
    {
        super(x, y);
    }

    /**
     * Constructeur valué à partir de deux points : construit le vecteur reliant
     * p1 à p2
     * @param p1 le premier point du vecteur
     * @param p2 le second point du vecteur
     */
    public Vecteur2D(Point2D p1, Point2D p2)
    {
        super(p2.x - p1.x, p2.y - p1.y);
    }

    /**
     * Calcul du produit scalaire avec un autre vecteur
     * @param v l'autre vecteur avec lequel calculer le produit scalaire
     * @return le produit scalaire du vecteur courant avec l'autre vecteur
     */
    public double dotProduct(Vecteur2D v)
    {
        return (x * v.x) + (y * v.y);
    }

    /**
     * Calcul de la norme du produit vectoriel avec un autre vecteur
     * @param v l'autre vecteur avec lequel calculer le produit scalaire
     * @return le produit scalaire du vecteur courant avec l'autre vecteur
     */
    public double crossProductN(Vecteur2D v)
    {
        return (x * v.y) - (y * v.x);
    }

    /**
     * Norme du vecteur
     * @return la norme du vecteur
     */
    public double norme()
    {
        return Math.sqrt(dotProduct(this));
    }

    /**
     * Normalisation d'un vecteur
     * @return renvoie le vecteur unitaire correspondant au vecteur
     */
    public Vecteur2D normalize()
    {
        double norme = norme();
        return new Vecteur2D(x/norme, y/norme);
    }
}
```



```java
/**
 * Package contenant l'ensemble des tests JUnit4
 */
package tests;
```

```java
package figures;

import points.Point2D;

/**
 * Interface des figures
 * @author davidroussel
 */
public interface Figure extends Cloneable
{
    /**
     * Accesseur en lecture pour le nom de la figure
     * @return une chaine contenant le nom de la figure
     */
    public abstract String getNom();

    /**
     * Méthode abstraite
     * Déplacement de la figure
     * @param dx déplacement selon l'axe des X
     * @param dy déplacement selon l'axe des Y
     * @return renvoie une référence vers la figure afin que l'on puisse déplacer une
     * figure en cascade  : <code>f.deplace(dx,dy).deplace(dx,dy)</code>
     * @see Point2D#deplace(double, double)
     */
    public abstract Figure deplace(double dx, double dy);

    /**
     * Affichage contenu
     * @return une chaine de caractère représentant la figure
     */
    @Override
    public abstract String toString();

    /**
     * Test de contenu d'un point dans la figure
     * teste si le point passé en argument est contenu à l'intérieur de la figure
     * @param p : point candidat à la contenance
     * @return la contenance du point à l'intérieur de la figure
     */
    public abstract boolean contient(Point2D p);

    /**
     * Centre de la figure.
     * renvoie le centre de la figure
     * @return renvoie le point2D central de la figure
     */
    public abstract Point2D getCentre();

    /**
     * Aire couverte par la figure
     * @return renvoie l'aire couverte par la figure
     */
    public abstract double aire();

    /**
     * Distance entre les centres de la figure courante et d'une figure
     * passée en argument
     * @param f figure avec laquelle on calcule la distance entre les centres
     * @return la distance entre les points centraux des deux figures
     * @see #getCentre()
     * @see Point2D#distance(Point2D, Point2D)
     */
    public default double distanceToCentreOf(Figure f)
    {
        // getCentre est une méthode abstraite mais rien ne nous empêche
        // de l'utiliser dans une autre méthode. Grâce au lien dynamique
        // TODO Remplacer par l'implémentation
        return 0.0;
    }

    /**
     * Distance entre les centres de deux figures
     * @param f1 première figure
     * @param f2 seconde figure
     * @return la distance entre les points centraux des deux figures
     * @see #getCentre()
     * @see Point2D#distance(Point2D, Point2D)
     */
    public static double distanceToCentre(Figure f1, Figure f2)
    {
        // getCentre est une méthode abstraite mais rien ne nous empêche
        // de l'utiliser dans une autre méthode. Grâce au lien dynamique
        // TODO Remplacer par l'implémentation
        return 0.0;
    }

    /**
     * Test d'égalité de la figure courante avec une autre figure.
     * Cette méthode n'implémente que le test sur la nature des figures,
```



```java
     * le test sur le contenu doit être réimplémenté dans chaque sous classe,
     * en utilisant cette méthode pour tester la nature des figures.
     * @param o la figure dont il faut comparer le contenu.
     * @return true si les deux figures sont de nature identique et qu'elles ont
     * le même contenu.
     */
    @Override
    public abstract boolean equals(Object o);
}
```

```java
package figures;
import points.Point2D;

/**
 * Classe abstraite Figure Contient une données concrête : le nom de la figure (
 * {@link #nom} )
 * <ul>
 * <li>des méthodes d'instance</li>
 * <ul>
 * <li>concrètes
 * <ul>
 * <li>un constructeur avec un nom : {@link #AbstractFigure(String)}</li>
 * <li>un accesseur pour ce nom : {@link #getNom()}</li>
 * <li>la méthode toString pour afficher ce nom {@link #toString()}</li>
 * <li> {@link #distanceToCentreOf(Figure)}</li>
 * </ul>
 * <li>abstraites
 * <ul>
 * <li> {@link #deplace(double,double)}</li>
 * <li> {@link #contient(Point2D)}</li>
 * <li> {@link #getCentre()}</li>
 * <li> {@link #aire()}</li>
 * </ul>
 * </ul>
 * <li>des méthodes de classes</li>
 * <ul>
 * <li>concrètes</li>
 * <ul>
 * <li> {@link #distanceToCentre(Figure,Figure)}</li>
 * </ul>
 * </ul> </ul>
 *
 * @author David Roussel
 */
public abstract class AbstractFigure implements Figure
{
    /**
     * Nom de la figure
     */
    protected String nom;

    /**
     * Constructeur (protégé) par défaut.
     * Affecte le nom de la classe comme nom de figure
     */
    protected AbstractFigure()
    {
        nom = getClass().getSimpleName();
    }

    /**
     * Constructeur (protégé) avec un nom
     * on a fait exprès de ne pas mettre de constructeur sans arguments
     * @param unNom Chaine de caractère pour initialiser le nom de la
     * figure
     */
    protected AbstractFigure(String unNom)
    {
        nom = unNom;
    }

    /**
     * @return le nom
     * @see figures.Figure#getNom()
     */
    @Override
    public String getNom()
    {
        return nom;
    }

    /* (non-Javadoc)
     * @see figures.Figure#deplace(double, double)
     */
    @Override
    public abstract Figure deplace(double dx, double dy);

    /* (non-Javadoc)
     * @see figures.Figure#toString()
     */
    @Override
    public String toString()
    {
        return(nom + ":");
    }

    /* (non-Javadoc)
     * @see figures.Figure#contient(points.Point2D)
     */
    @Override
```



```java
    public abstract boolean contient(Point2D p);

    /* (non-Javadoc)
     * @see figures.Figure#getCentre()
     */
    @Override
    public abstract Point2D getCentre();

    /* (non-Javadoc)
     * @see figures.Figure#aire()
     */
    @Override
    public abstract double aire();

    /**
     * Comparaison de deux figures en termes de contenu
     * @return true si f est du même types que la figure courante et qu'elles
     * ont un contenu identique
     */
    protected abstract boolean equals(Figure f);

    /**
     * Comparaison de deux figures, on ne peut pas vérifier grand chose pour
     * l'instant à part la classe et le nom
     * @note implémentation partielle qui ne vérifie que null/this/et l'égalité
     * de classe
     * @see figures.Figure#equals(java.lang.Object)
     */
    @Override
    public boolean equals(Object obj)
    {
        // TODO remplacer par l'implémetation
        return false;
    }

//  /**
//   * Hashcode d'une figure (implémentation partielle basée sur le nom d'une
//   * figure) --> Non utilisé
//   * @see java.lang.Object#hashCode()
//   */
//  @Override
//  public int hashCode()
//  {
//      final int prime = 31;
//      int result = 1;
//      result = (prime * result) + ((nom == null) ? 0 : nom.hashCode());
//      return result;
//  }
}
```

```java
package figures;
import points.Point2D;
import points.Vecteur2D;

/**
 * Classe triangle héritière de la classe abstraite Figure le triangle est
 * composé de trois points doit donc implémenter les méthodes abstraites
 * suivantes
 *
 * @see AbstractFigure#deplace
 * @see AbstractFigure#contient
 * @see AbstractFigure#getCentre
 * @see AbstractFigure#aire
 */
public class Triangle extends AbstractFigure
{
    /**
     * tableau de 3 points
     *
     * @uml.property name="points"
     * @uml.associationEnd multiplicity="(0 -1)" dimension="1" ordering="true"
     *                     aggregation="composite"
     *                     inverse="triangle:points.Point2D"
     */
    protected Point2D[] points = new Point2D[3];

    // Constructeurs ------------------------------------------------------
    /**
     * Constructeur par défaut : construit un triangle isocèle de 1 de base
     *  et de 1 de haut à partir de l'origine
     */
    public Triangle()
    {
        points[0] = new Point2D(0.0, 0.0);
        points[1] = new Point2D(1.0, 0.0);
        points[2] = new Point2D(0.5, 1.0);
    }

    /**
     * Constructeur valué : construit un triangle à partir de 3 points
     * @param p1 premier point
     * @param p2 second point
     * @param p3 troisième point
     */
    public Triangle(Point2D p1, Point2D p2, Point2D p3)
    {
        points[0] = new Point2D(p1);
        points[1] = new Point2D(p2);
        points[2] = new Point2D(p3);
    }

    /**
     * Constructeur de copie.
     * @param t le triangle à copier.
     */
    public Triangle(Triangle t)
    {
        this(t.points[0], t.points[1], t.points[2]);
    }


    // Accessseurs --------------------------------------------------------
    /**
     * Accesseur en lecture pour le n<sup><font size=\"-2\">ième</font></sup>
     * point (avec n dans [0..2])
     * @param n l'indice du point recherché
     * @return le n<sup><font size=\"-2\">ième</font></sup> point du triangle
     */
    public Point2D getPoint(int n)
    {
        if ( (n > (points.length - 1)) ∨ (n < 0) )
        {
            System.err.println("Triangle getPoint index invalide");
            return null;
        }
        else
        {
            return points[n];
        }
    }

    // Implémentation de Figure -------------------------------------------
    /**
     * Implementation Figure,
     * Déplacement du triangle
     * @param dx déplacement suivant x
     * @param dy déplacement suivant y
     * @return une référence vers la figure déplacée
     */
    @Override
```



```java
    public Figure deplace(double dx, double dy)
    {
        for (Point2D p : points)
        {
            p.deplace(dx, dy);
        }
        return this;
    }

    /**
     * Implementation Figure,
     * Affichage contenu
     * @return une chaine représentant l'objet (les trois points)
     */
    @Override
    public String toString()
    {
        StringBuilder result = new StringBuilder(super.toString());

        for (int i=0; i < points.length; i++)
        {
            result.append(points[i].toString());

            if (i < (points.length - 1))
            {
                result.append(",");
            }
        }

        return result.toString();
    }

    /**
     * Test de contenu : teste si le point passé en argument est contenu à
     * l'intérieur du triangle.
     * Pour savoir si un point est contenu dans un polygone convexe
     * il suffit d'effectuer le produit vectoriel des vecteurs
     * reliant ce point avec deux points consécutifs le long du
     * polygone, et ceci le long de chaque paire de points dans le
     * polygone.
     * Si on observe un changement de signe du produit vectoriel entre
     * deux paires de vecteurs cela signifie que le point se trouve à
     * l'extérieur du polygone.
     * Contre-exemple : lorsqu'un point se trouve à l'intérieur d'un
     * polygone convexe la suite des produits vectoriels des paires de
     * vecteurs ne change jamais de signe !
     * @param p point à tester
     * @return une valeur booléenne indiquant si le point est contenu ou pas
     * à l'intérieur du triangle
     */
    @Override
    public boolean contient(Point2D p)
    {

        // Résultat initial
        boolean result = true;

        // Vecteurs initiaux
        Vecteur2D v1 = new Vecteur2D(p,points[0]);
        Vecteur2D v2 = new Vecteur2D(p,points[1]);

        // premier produit vectoriel
        double crossp = v1.crossProductN(v2);

        // signe produit vectoriel initial
        double signInit = crossp ≥ 0 ? 1 : -1;

        // produits vectoriels suivants
        double sign;

        // parcours des points du polygone à la recherche d'un changement
        // de signe du produit vectoriel
        for (int i=1; i<points.length; i++)
        {
            v1 = v2;
            v2 = new Vecteur2D(p, points[(i+1)%points.length]);

            crossp = v1.crossProductN(v2);
            sign = crossp ≥ 0 ? 1 : -1;

            if (sign ≠ signInit)
            {
                result = false;
                break;
            }
        }

        return result;
    }

```

```
181     /**
182      * Accesseur en lecture du centre de masse du triangle ( = barycentre)
183      * @return renvoie le barycentre du triangle
184      */
185     @Override
186     public Point2D getCentre()
187     {
188         double sx = 0.0;
189         double sy = 0.0;
190         // somme des coordonnées des points
191         for (int i=0; i<points.length; i++)
192         {
193             sx+=points[i].getX();
194             sy+=points[i].getY();
195         }
196         // renvoi de la moyenne de chaque coordonnée
197         return new Point2D(sx/points.length, sy/points.length);
198     }
199 
200     /**
201      * Calcul de l'aire d'un triangle
202      * @return l'aire couverte par le triangle
203      */
204     @Override
205     public double aire()
206     {
207         // pour calculer l'aire d'un polygone convexe du plan XY, on utilise
208         // une nouvelle fois les propriétés du produit vectoriel.
209         // La norme du produit vectoriel représente le double de l'aire
210         // couverte par les deux vecteurs dont on calcule ce produit.
211         // il suffit donc de faire cette somme sur tous les triangle qui
212         // composent le polygone en formant des vecteurs constitués par des
213         // couples de points consécutifs le long du polygone.
214         // Bon tout ca c'est bien mais pour un triangle c'est plus simple :
215 
216         Vecteur2D v1 = new Vecteur2D(points[0], points[1]);
217         Vecteur2D v2 = new Vecteur2D(points[0], points[2]);
218 
219         return (Math.abs(v1.crossProductN(v2)) / 2.0);
220     }
221 
222     /**
223      * Comparaison de deux triangles. On considère que deux triangles sont
224      * identiques s'ils contiennent les mêmes points (pas forcément dans
225      * le même ordre)
226      * @see Figure#equals(java.lang.Object)
227      */
228     @Override
229     public boolean equals(Figure figure)
230     {
231         if (getClass().equals(figure.getClass()))
232         {
233             Triangle other = (Triangle) figure;
234             for (int i = 0; i < points.length; i++)
235             {
236                 boolean found = false;
237                 for (int j = 0; j < points.length; j++)
238                 {
239                     if (points[i].equals(other.points[j]))
240                     {
241                         found = true;
242                         break;
243                     }
244                 }
245                 if (¬found)
246                 {
247                     return false;
248                 }
249             }
250             return true;
251         }
252         else
253         {
254             return false;
255         }
256     }
257 }
```



```
1  /**
2   * Package contenant l'ensemble des tests JUnit4
3   */
4  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(
{
    ListeTest.class,
    CollectionListeTest.class,
    Point2DTest.class,
    FigureTest.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.assertNotSame;
import static org.junit.Assert.assertSame;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;

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

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

import listes.Liste;

/**
 * Classe de test de la liste Chainée
 * @author davidroussel
 */
public class ListeTest
{

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

    /**
     * Liste des éléments à insérer dans la liste
     */
    private static String[] elements;

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

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

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

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

```
91      }
92
93      /**
94       * Méthode utilitaire de remplissage de la liste avec les éléments
95       * du tableau #elements
96       */
97      private final void remplissage()
98      {
99          if (liste ≠ null)
100         {
101             for (String elt : elements)
102             {
103                 liste.add(elt);
104             }
105         }
106     }
107
108     /**
109      * Test method for {@link listes.Liste#Liste()}.
110      */
111     @Test
112     public final void testListe()
113     {
114         String testName = new String("Liste<String>()");
115         System.out.println(testName);
116
117         assertNotNull(testName + " instance non null failed", liste);
118         assertTrue(testName + " liste vide failed", liste.empty());
119     }
120
121     /**
122      * Test method for {@link listes.Liste#Liste(listes.Liste)}.
123      */
124     @Test
125     public final void testListeListeOfT()
126     {
127         String testName = new String("Liste<String>(Liste<String>)");
128         System.out.println(testName);
129
130         Liste<String> liste2 = new Liste<String>();
131         liste = new Liste<String>(liste2);
132
133         assertNotNull(testName + " instance non null failed", liste);
134         assertTrue(testName + " liste vide failed", liste.empty());
135
136         remplissage();
137         assertFalse(testName + " liste remplie failed", liste.empty());
138         liste2 = new Liste<String>(liste);
139         assertNotNull(testName + " copie liste remplie failed", liste2);
140         assertEquals(testName + " contenus égaux failed", liste, liste2);
141     }
142
143     /**
144      * Test method for {@link listes.Liste#add(java.lang.Object)}.
145      */
146     @Test
147     public final void testAdd()
148     {
149         String testName = new String("Liste<String>.add(E)");
150         System.out.println(testName);
151
152         // Ajout dans une liste vide
153         liste.add(elements[0]);
154         assertFalse(testName + "liste non vide failed", liste.empty());
155         Iterator<String> it = liste.iterator();
156         String insertedElt = it.next();
157         assertSame(testName + " contrôle ref element[0] failed", insertedElt, elements[0]);
158         // Si assertSame réussit asserEquals n'est plus nécessaire
159
160         // Ajout dans une liste non vide
161         for (int i=1; i < elements.length; i++)
162         {
163             liste.add(elements[i]);
164             /*
165              * Attention le précédent "it" a été invalidé par l'ajout
166              * Lors du dernier next le current de l'itérateur est passé à null
167              * puisqu'il n'y avait pas (encore) de suivant, donc retenter un
168              * next sur le même itérateur génèrera un NoSuchelementException.
169              * Il faut donc réobtenir un itérateur pour parcourir la liste
170              * après un ajout
171              */
172             it = liste.iterator();
173             for (int j = 0; j ≤ i; j++)
174             {
175                 insertedElt = it.next();
176             }
177             assertSame(testName + " contrôle ref element[" + i + "] failed",
178                     insertedElt, elements[i]);
179         }
180     }
```



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

```
271         int index = 0;
272
273         // - insertion d'un élément 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 début de liste non vide avec un index valide
302         result = liste.insert(elements[nextIndex[index]], 0);
303         // liste = Hello -> Brave ->
304         assertTrue(testName + " insertion début 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 élément 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 élement 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 élément null d'une liste non vide
346         result = liste.remove(null);
347         assertFalse(testName + " null failed", result);
348
349         // suppression d'un élément inexistant d'une liste non vide
350         result = liste.remove("Coucou");
351         assertFalse(testName + " Coucou failed", result);
352
353         // suppression d'un élement existant en début 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 élement 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 élement 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 élément 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 élement 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 élément 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 début, au milieu et à 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-même
528         result = liste.equals(liste);
529         assertTrue(testName + " self failed", result);
530
531         // Egalite sur liste copiée
532         Liste<String> liste2 = new Liste<String>(liste);
533         result = liste.equals(liste2);
534         assertTrue(testName + " copy failed", result);
535
536         // Inegalité sur listes de tailles différentes
537         liste2.add("of Pain");
538         result = liste.equals(liste2);
539         assertFalse(testName + " copy + of Pain failed", result);
540
```



```
541         // Inegalite sur liste à 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 même contenu
551         // SSI equals compare un Iterable plutôt 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'itérateur
599         }
600
601         assertFalse(testName + " finished", it.hasNext());
602
603         // Un appel supplémentaire à next sur un itérateur terminé
604         // doit soulever une NoSuchElementException
605         it.next();
606
607         fail(testName + " next sur itérateur terminé");
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 à 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.assertSame;
7  import static org.junit.Assert.assertTrue;
8  import static org.junit.Assert.fail;
9
10 import java.util.ArrayList;
11 import java.util.Collection;
12 import java.util.Iterator;
13
14 import org.junit.After;
15 import org.junit.AfterClass;
16 import org.junit.Before;
17 import org.junit.BeforeClass;
18 import org.junit.Test;
19
20 import listes.CollectionListe;
21
22 /**
23  * Classe de test de la CollectionListe en tant que Collection
24  *
25  * @author davidroussel
26  */
27 public class CollectionListeTest
28 {
29     /**
30      * La liste à tester. La nature du contenu de la liste importe peu du moment
31      * qu'il est homogène : donc n'importe quel type ferait l'affaire.
32      */
33     private CollectionListe<String> collection;
34
35     /**
36      * Liste des éléments à ajouter à la collection
37      */
38     private static String[] elements = new String[] {
39         "Hello",
40         "Brave",
41         "New",
42         "World" };
43
44     /**
45      * Element supplémentaire à ajouter à la collection
46      */
47     private static String extraElement = new String("Of Pain");
48
49     /**
50      * Mise en place avant l'ensemble des tests
51      *
52      * @throws java.lang.Exception
53      */
54     @BeforeClass
55     public static void setUpBeforeClass() throws Exception
56     {
57         // rien
58     }
59
60     /**
61      * Nettoyage après l'ensemble des tests
62      *
63      * @throws java.lang.Exception
64      */
65     @AfterClass
66     public static void tearDownAfterClass() throws Exception
67     {
68         // rien
69     }
70
71     /**
72      * Mise en place avant chaque test
73      *
74      * @throws java.lang.Exception
75      */
76     @Before
77     public void setUp() throws Exception
78     {
79         collection = new CollectionListe<String>();
80     }
81
82     /**
83      * Nettoyage après chaque test
84      *
85      * @throws java.lang.Exception
86      */
87     @After
88     public void tearDown() throws Exception
89     {
90         collection.clear();
```

```
91          collection = null;
92          System.gc();
93      }
94
95      /**
96       * Remplissage d'une collection avec les éléments de #elements
97       * @param collection la collection à remplir
98       */
99      public static void remplissage(Collection<String> collection)
100     {
101         for (String elt : elements)
102         {
103             collection.add(elt);
104         }
105     }
106
107     /**
108      * Test method for {@link listes.CollectionListe#CollectionListe()}.
109      */
110     @Test
111     public final void testCollectionListe()
112     {
113         String testName = new String("CollectionListe<String>()");
114         System.out.println(testName);
115
116         assertNotNull(testName + " instance", collection);
117         assertTrue(testName + " empty", collection.isEmpty());
118         assertEquals(testName + " size == 0", 0, collection.size());
119     }
120
121     /**
122      * Test method for
123      * {@link listes.CollectionListe#CollectionListe(java.util.Collection)}.
124      */
125     @Test
126     public final void testCollectionListeCollectionOfE()
127     {
128         String testName = new String(
129                 "CollectionListe<String>(Collection<String>)");
130         System.out.println(testName);
131
132         ArrayList<String> otherCollection = new ArrayList<String>();
133         remplissage(otherCollection);
134
135         collection = new CollectionListe<String>(otherCollection);
136
137         assertNotNull(testName + " instance", collection);
138         assertFalse(testName + " not empty", collection.isEmpty());
139         assertEquals(testName + " size", elements.length, collection.size());
140         int i = 0;
141         for (String elt : collection)
142         {
143             assertSame(testName + " elt[" + String.valueOf(i) + "]",
144                     elements[i++], elt);
145         }
146     }
147
148     /**
149      * Test method for {@link listes.CollectionListe#add(java.lang.Object)}.
150      */
151     @Test
152     public final void testAddE()
153     {
154         String testName = new String("CollectionListe<String>.add(String)");
155         System.out.println(testName);
156
157         collection.add(extraElement);
158
159         assertEquals(testName + " size", 1, collection.size());
160         Iterator<String> it = collection.iterator();
161         assertTrue(testName + " iterator not empty", it.hasNext());
162         assertSame(testName + " element", extraElement, it.next());
163         assertFalse(testName + " iterator end", it.hasNext());
164     }
165
166     /**
167      * Test method for
168      * {@link java.util.AbstractCollection#addAll(java.util.Collection)}.
169      */
170     @Test
171     public final void testAddAll()
172     {
173         String testName = new String(
174                 "CollectionListe<String>.addAll(Collection<String>)");
175         System.out.println(testName);
176
177         ArrayList<String> otherCollection = new ArrayList<String>();
178         remplissage(otherCollection);
179
180         collection.addAll(otherCollection);
```



```
181
182         assertNotNull(testName + " instance", collection);
183         assertFalse(testName + " not empty", collection.isEmpty());
184         assertEquals(testName + " size", elements.length, collection.size());
185         int i = 0;
186         for (String elt : collection)
187         {
188             assertSame(testName + " elt[" + String.valueOf(i) + "]",
189                     elements[i++], elt);
190         }
191     }
192
193     /**
194      * Test method for {@link java.util.AbstractCollection#clear()}.
195      */
196     @Test
197     public final void testClear()
198     {
199         String testName = new String("CollectionListe<String>.clear()");
200         System.out.println(testName);
201         boolean result;
202
203         // Remplissage
204         remplissage(collection);
205
206         // Non vide après remplissage
207         result = collection.isEmpty();
208         assertFalse(testName + " rempli", result);
209
210         collection.clear();
211
212         // Vide après clear
213         result = collection.isEmpty();
214         assertTrue(testName + " effacé", result);
215     }
216
217     /**
218      * Test method for
219      * {@link java.util.AbstractCollection#contains(java.lang.Object)}.
220      */
221     @Test
222     public final void testContains()
223     {
224         String testName = new String("CollectionListe<String>.Contains(String)");
225         System.out.println(testName);
226         boolean result;
227
228         // Recherche contenu null sur une collection vide
229         result = collection.contains(null);
230         assertFalse(testName + "null sur col vide", result);
231
232         // Recherche contenu non null sur une collection vide
233         result = collection.contains("Bonjour");
234         assertFalse(testName + " non null sur col vide", result);
235
236         // Remplissage
237         remplissage(collection);
238
239         // Contenu null non trouvé sur liste remplie
240         result = collection.contains(null);
241         assertFalse(testName + "null sur col remplie", result);
242
243         // Recherche contenu non null non contenu sur une collection remplie
244         result = collection.contains("Bonjour");
245         assertFalse(testName + " non null sur col remplie", result);
246
247         for (String elt : elements)
248         {
249             // Recherche contenu non null contenu dans collection remplie
250             result = collection.contains(elt);
251             assertTrue(testName + " non null sur col remplie", result);
252         }
253     }
254
255     /**
256      * Test method for
257      * {@link java.util.AbstractCollection#containsAll(java.util.Collection)}.
258      */
259     @Test
260     public final void testContainsAll()
261     {
262         String testName = new String(
263                 "CollectionListe<String>.ContainsAll(Collection<String>)");
264         System.out.println(testName);
265         boolean result;
266
267         // Recherche contenu null sur une collection vide
268         try
269         {
270             result = collection.containsAll(null);
```

```
271
272             fail(testName + " null sur collection vide sans exception");
273         }
274         catch (NullPointerException npe)
275         {
276             // il est normal d'obtenir une telle exception donc rien
277         }
278
279         // Remplissage autre collection dans l'ordre direct
280         ArrayList<String> forwardCollection = new ArrayList<String>();
281         remplissage(forwardCollection);
282
283         // Ajout dans autre collection directe d'un elt supplémentaire
284         ArrayList<String> forwardCollectionPlus = new ArrayList<String>(
285                 forwardCollection);
286         forwardCollectionPlus.add(extraElement);
287
288         // Recherche contenu non null sur une collection vide
289         result = collection.containsAll(forwardCollection);
290         assertFalse(testName + " non null sur col vide", result);
291
292         // Remplissage autre collection dans l'ordre inverse
293         ArrayList<String> reverseCollection = new ArrayList<String>();
294         for (int i = elements.length - 1; i ≥ 0; i--)
295         {
296             reverseCollection.add(elements[i]);
297         }
298         // Ajout dans autre collection inverse d'un elt supplémentaire
299         ArrayList<String> reverseCollectionPlus = new ArrayList<String>(
300                 reverseCollection);
301         reverseCollectionPlus.add(extraElement);
302
303         // Remplissage autre collection différente
304         ArrayList<String> otherCollection = new ArrayList<String>();
305         otherCollection.add("Bonjour");
306         otherCollection.add("Brave");
307         otherCollection.add("Nouveau");
308         otherCollection.add("Monde");
309
310         // Remplissage collection
311         remplissage(collection);
312
313         CollectionListe<String> collectionPlus = new CollectionListe<String>(
314                 collection);
315         collectionPlus.add(extraElement);
316
317         // Contenu null non trouvé sur liste remplie
318         try
319         {
320             result = collection.containsAll(null);
321
322             fail(testName + "null sur col remplie sans exception");
323         }
324         catch (NullPointerException npe)
325         {
326             // il est normal d'obtenir une telle exception donc rien
327         }
328
329         // Recherche contenu non null non contenu sur une collection remplie
330         result = collection.containsAll(otherCollection);
331         assertFalse(testName + " non null sur col remplie", result);
332
333         // Recherche contenu identique
334         result = collection.containsAll(forwardCollection);
335         assertTrue(testName + " identique sur col remplie", result);
336         result = collection.containsAll(reverseCollection);
337         assertTrue(testName + " inversé sur col remplie", result);
338
339         // Recherche contenu plus petit
340         result = collectionPlus.containsAll(forwardCollection);
341         assertTrue(testName + " plus petit identique sur col remplie", result);
342         result = collectionPlus.containsAll(reverseCollection);
343         assertTrue(testName + " plus petit inversé sur col remplie", result);
344
345         // Recherche contenu plus grand
346         result = collection.containsAll(forwardCollectionPlus);
347         assertFalse(testName + " plus grand identique sur col remplie", result);
348         result = collection.containsAll(reverseCollectionPlus);
349         assertFalse(testName + " plus petit inversé sur col remplie", result);
350     }
351
352     /**
353      * Test method for {@link java.util.AbstractCollection#isEmpty()}.
354      */
355     @Test
356     public final void testIsEmpty()
357     {
358         String testName = new String("CollectionListe<String>.isEmpty()");
359         System.out.println(testName);
360         boolean result = collection.isEmpty();
```



```
361
362         assertTrue(testName + " vide", result);
363
364         // Remplissage
365         remplissage(collection);
366
367         result = collection.isEmpty();
368         assertFalse(testName + " non vide", result);
369     }
370
371     /**
372      * Test method for {@link listes.CollectionListe#iterator()}.
373      */
374     @Test
375     public final void testIterator()
376     {
377         String testName = new String("CollectionListe<String>.iterator()");
378         System.out.println(testName);
379
380         // Itérateur sur liste vide
381         Iterator<String> result = collection.iterator();
382
383         assertNotNull(testName + " iterator non null", result);
384         assertFalse(testName + " iterator vide", result.hasNext());
385
386         // Remplissage
387         remplissage(collection);
388
389         // Itérateur sur liste non vide
390         result = collection.iterator();
391         assertNotNull(testName + " iterator non null", result);
392         assertTrue(testName + " iterator vide", result.hasNext());
393
394         for (int i = 0; i < elements.length; i++)
395         {
396             assertSame(testName + " iteration[" + String.valueOf(i) + "]",
397                     elements[i], result.next());
398         }
399         assertFalse(testName + " iterator terminé", result.hasNext());
400     }
401
402     /**
403      * Test method for
404      * {@link java.util.AbstractCollection#remove(java.lang.Object)}.
405      */
406     @Test
407     public final void testRemove()
408     {
409         String testName = new String("CollectionListe<String>.remove(String)");
410         System.out.println(testName);
411
412         // Retrait d'un élément null sur collection vide
413         boolean result = collection.remove(null);
414         assertFalse(testName + " retrait elt null sur col vide", result);
415
416         // Retrait d'un élement non null sur collection vide
417         result = collection.remove("Bonjour");
418         assertFalse(testName + " retrait elt sur col vide", result);
419
420         // Double Remplissage (pour vérifier l'ordre des retraits)
421         remplissage(collection);
422         remplissage(collection);
423         // collection = Hello -> Brave -> New -> World -> Hello -> Brave -> New -> World
424
425         // Retrait d'un élément null sur collection remplie
426         result = collection.remove(null);
427         assertFalse(testName + " retrait elt null sur col", result);
428
429         for (String elt : elements)
430         {
431             // retrait de la première occurrence
432             result = collection.remove(elt);
433             // la seconde occurrence est toujours présence
434             assertTrue(testName + " retrait 1ere occurrence", result);
435             assertTrue(testName + " persistence 2eme occurrence",
436                     collection.contains(elt));
437
438             // retrait de la seconde occurrence
439             result = collection.remove(elt);
440             assertTrue(testName + " retrait 2nde occurrence", result);
441             assertFalse(testName + " absence 2eme occurrence",
442                     collection.contains(elt));
443
444             // retrait elt non présent
445             result = collection.remove(elt);
446             assertFalse(testName + " retrait elt non présent", result);
447         }
448     }
449
450     /**
```

```
451      * Test method for
452      * {@link java.util.AbstractCollection#removeAll(java.util.Collection)}.
453      */
454     @Test
455     public final void testRemoveAll()
456     {
457         String testName = new String("CollectionListe<String>.removeAll(" +
458                 "Collection<String>)");
459         System.out.println(testName);
460         boolean result;
461 
462         // Retrait collection nulle sur collection vide
463         // Devrait générer un exception
464         try
465         {
466             result = collection.removeAll(null);
467 
468             fail(testName + " retrait collection null sur collection vide " +
469                     "sans exception");
470         }
471         catch (NullPointerException npe)
472         {
473             // Rien, on s'attends à cette exception
474         }
475 
476         // Double Remplissage autre collection
477         ArrayList<String> otherCollection = new ArrayList<String>();
478         remplissage(otherCollection);
479         remplissage(otherCollection);
480 
481         // Retrait othercollection sur collection vide
482         result = collection.removeAll(otherCollection);
483         assertFalse(testName + " retrait collection sur collection vide", result);
484 
485         // Remplissage collection
486         remplissage(collection);
487 
488         // Retrait collection nulle sur collection remplie
489         try
490         {
491             result = collection.removeAll(null);
492 
493             fail(testName +   " retrait collection null sur collection remplie" +
494                     " sans exception");
495         }
496         catch (NullPointerException npe)
497         {
498             // Rien, on s'attends à cette exception
499         }
500 
501         // Retrait otherCollection de collection même taille
502         result = collection.removeAll(otherCollection);
503         assertTrue(testName + " retrait collection +", result);
504         result = collection.isEmpty();
505         assertTrue(testName + " collection vide après retrait collection +",
506                 result);
507 
508         // Re-remplissages
509         otherCollection.clear();
510         remplissage(collection);
511         remplissage(otherCollection);
512 
513         CollectionListe<String> collectionPlus = new CollectionListe<String>(
514                 collection);
515         collectionPlus.add(extraElement);
516 
517         // Retrait collection plus grande
518         result = collection.removeAll(collectionPlus);
519         assertTrue(testName + " retrait collection plus grande", result);
520         assertTrue(testName + " col vide après retrait collection plus grande",
521                 collection.isEmpty());
522 
523         // Retrait collection plus petite
524         result = collectionPlus.removeAll(otherCollection);
525         assertTrue(testName + " retrait collection plus petite", result);
526         assertEquals(testName + " taille 1 après retrait collection plus " +
527                 "petite", 1, collectionPlus.size());
528     }
529 
530     /**
531      * Test method for
532      * {@link java.util.AbstractCollection#retainAll(java.util.Collection)}.
533      */
534     @Test
535     public final void testRetainAll()
536     {
537         String testName = new String("CollectionListe<String>.retainAll(" +
538                 "Collection<String>)");
539         System.out.println(testName);
540         boolean result;
```



```
541 
542         // Retain collection null sur collection vide
543         // Devrait générer une exception
544         try
545         {
546             result = collection.retainAll(null);
547             fail(testName + " retainAll(null) sur collection vide sans " +
548                     "exception");
549         }
550         catch (NullPointerException npe)
551         {
552             // Rien, on s'attends à cette exception
553         }
554 
555         // Remplissage otherCollection
556         ArrayList<String> otherCollection = new ArrayList<String>();
557         remplissage(otherCollection);
558 
559         // Retain otherCollection sur collection vide
560         result = collection.retainAll(otherCollection);
561         assertFalse(testName + " retainAll elements sur colelction vide", result);
562 
563         // Remplissage collection
564         collection.addAll(otherCollection);
565         collection.add(extraElement);
566 
567         // Retain null collection sur collection remplie
568         try
569         {
570             result = collection.retainAll(null);
571 
572             fail(testName + " retainAll(null) sur collection remplie sans " +
573                     "exception");
574         }
575         catch (NullPointerException npe)
576         {
577             // Rien, on s'attends à cette exception
578         }
579 
580         // Retain otherCollection sur collection remplie + extra element
581         result = collection.retainAll(otherCollection);
582         assertTrue(testName + " retainAll(other) sur col. remplie+", result);
583         assertEquals(testName + " retainAll(other) sur col. remplie+ size",
584                 otherCollection.size(), collection.size());
585         Iterator<String> it1 = collection.iterator();
586         Iterator<String> it2 = otherCollection.iterator();
587         for (; it1.hasNext() ∧ it2.hasNext();)
588         {
589             assertSame(testName + " retainAll test same elts", it1.next(),
590                     it2.next());
591         }
592     }
593 
594     /**
595      * Test method for {@link listes.CollectionListe#size()}.
596      */
597     @Test
598     public final void testSize()
599     {
600         String testName = new String("CollectionListe<String>.size()");
601         System.out.println(testName);
602         int result;
603 
604         // Taille nulle sur collection vide
605         result = collection.size();
606         assertEquals(testName + " taille nulle sur collection vide", 0, result);
607 
608         // Remplissage
609         remplissage(collection);
610 
611         // Taille  après remplissage
612         result = collection.size();
613         assertEquals(testName + " taille collection après remplissage",
614                 elements.length, result);
615     }
616 
617     /**
618      * Test method for {@link java.util.AbstractCollection#toArray()}.
619      */
620     @Test
621     public final void testToArray()
622     {
623         String testName = new String("CollectionListe<String>.toArray()");
624         System.out.println(testName);
625         Object[] result;
626 
627         // toArray sur collection vide
628         result = collection.toArray();
629         assertEquals(testName + " toArray collection vide", 0, result.length);
630 
```

```
631         // Remplissage
632         remplissage(collection);
633 
634         // toArray après remplissage
635         result = collection.toArray();
636         assertEquals(testName + " toArray après remplissage",
637                 elements.length, result.length);
638         for (int i = 0; i < elements.length; i++)
639         {
640             assertSame(testName + " element[" + String.valueOf(i) + "]",
641                     elements[i], result[i]);
642         }
643     }
644 
645     /**
646      * Test method for {@link java.util.AbstractCollection#toArray(T[])}.
647      */
648     @Test
649     public final void testToArrayTArray()
650     {
651         String testName = new String("CollectionListe<String>.toArray(T[])");
652         System.out.println(testName);
653         String[] result;
654 
655         // toArray sur collection vide
656         result = collection.toArray(new String[0]);
657         assertEquals(testName + " collection vide", 0, result.length);
658 
659         // Remplissage
660         remplissage(collection);
661 
662         // toArray après remplissage
663         result = collection.toArray(new String[0]);
664         assertEquals(testName + " après remplissage",
665                 elements.length, result.length);
666         for (int i = 0; i < elements.length; i++)
667         {
668             assertSame(testName + " element[" + String.valueOf(i) + "]",
669                     elements[i], result[i]);
670         }
671     }
672 
673     /**
674      * Test method for {@link java.util.AbstractCollection#toString()}.
675      */
676     @Test
677     public final void testToString()
678     {
679         String testName = new String("CollectionListe<String>.toString()");
680         System.out.println(testName);
681         String result;
682 
683         // Remplissage
684         remplissage(collection);
685 
686         String expected = new String("[Hello, Brave, New, World]");
687 
688         result = collection.toString();
689 
690         assertEquals(testName, expected, result);
691     }
692 
693     /**
694      * Test method for {@link listes.CollectionListe#equals(java.lang.Object)}.
695      */
696     @Test
697     public final void testEqualsObject()
698     {
699         String testName = new String("CollectionListe<String>.equals(Object)");
700         System.out.println(testName);
701         boolean result;
702 
703         // Equals sur null
704         result = collection.equals(null);
705         assertFalse(testName + " null object", result);
706 
707         // Remplissage
708         remplissage(collection);
709 
710         // Equals sur this
711         result = collection.equals(collection);
712         assertTrue(testName + " this", result);
713 
714         // Equals sur objet de nature différente
715         result = collection.equals(new Object());
716         assertFalse(testName + " this", result);
717 
718         // Equals sur CollectionListe non semblable
719         CollectionListe<String> otherCollectionListe = new CollectionListe<String>(
720                 collection);
```



```
721         collection.add(extraElement);
722         result = collection.equals(otherCollectionListe);
723         assertFalse(testName + " otherCollectionListe non semblable", result);
724 
725         // Equals sur CollectionListe semblable
726         otherCollectionListe.add(extraElement);
727         result = collection.equals(otherCollectionListe);
728         assertTrue(testName + " otherCollectionListe semblable", result);
729 
730         // Equals sur Collection non semblable
731         collection.remove(extraElement);
732         ArrayList<String> otherCollection = new ArrayList<String>(collection);
733         collection.add(extraElement);
734         result = collection.equals(otherCollection);
735         assertFalse(testName + " otherCollection non semblable", result);
736 
737         // Equals sur Collection semblable
738         // CollectionListe<E> peut se comparer à toute Collection<E>
739         otherCollection.add(extraElement);
740         result = collection.equals(otherCollection);
741         assertTrue(testName + " equals direct", result);
742         // ArrayList<E> ne peut se comparer qu'à une autre List<E>
743         boolean resultInverse = otherCollection.equals(collection);
744         assertFalse(testName + " equals inverse", resultInverse);
745     }
746 
747     /**
748      * Test method for {@link listes.CollectionListe#hashCode()}.
749      */
750     @Test
751     public final void testHashCode()
752     {
753         String testName = new String("CollectionListe<String>.equals(Object)");
754         System.out.println(testName);
755         int result1, result2;
756 
757         ArrayList<String> otherCollection = new ArrayList<String>();
758 
759         // hashCode collection vide = 1
760         result1 = collection.hashCode();
761         result2 = otherCollection.hashCode();
762         assertEquals(testName + " hashCode collection vide", 1, result1);
763         assertEquals(testName + " hashCodes collections vides", result2, result1);
764 
765         // Remplissages
766         remplissage(collection);
767         remplissage(otherCollection);
768 
769         // hasCode collections semblables
770         result1 = collection.hashCode();
771         result2 = otherCollection.hashCode();
772         assertEquals(testName + " hashCode collections remplies", result2,
773                 result1);
774 
775         // hasCode collections dissemblables
776         collection.add(extraElement);
777         result1 = collection.hashCode();
778         assertTrue(testName + " hashCode collections remplies +",
779                 result2 ≠ result1);
780 
781         // [Optionnel]
782         // Les collections dissemblables ne sont plus égales
783         assertFalse(testName + " hashCode + equals direct +",
784                 collection.equals(otherCollection));
785     }
786 }
```

```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.assertSame;
import static org.junit.Assert.assertTrue;

import java.util.ArrayList;

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

import points.Point2D;

/**
 * Class de test de la classe {@link Point2D}
 *
 * @author David
 */
public class Point2DTest
{
    /**
     * Le point2D à tester
     */
    private Point2D point;

    /**
     * Liste de points
     */
    private ArrayList<Point2D> points;

    /**
     * Etendue max pour les random
     */
    private static final double maxRandom = 1e9;

    /**
     * Nombre d'essais pour les tests
     */
    private static final long nbTrials = 1000;

    /**
     * Nombre de subdivisions pour les étendues lors des tests
     */
    private static final int nbSteps = 100;

    /**
     * Constructeur de la classe de test. Initialise les attributs utilisés dans
     * les tests
     */
    public Point2DTest()
    {
        point = null;
        points = new ArrayList<Point2D>();
    }

    /**
     * Mise en place avant tous les tests
     *
     * @throws java.lang.Exception
     */
    @BeforeClass
    public static void setUpBeforeClass() throws Exception
    {
        // Rien
    }

    /**
     * Nettoyage après tous les tests
     *
     * @throws java.lang.Exception
     */
    @AfterClass
    public static void tearDownAfterClass() throws Exception
    {
        // Rien
    }

    /**
     * Mise en place avant chaque test
     *
     * @throws java.lang.Exception
     */
    @Before
    public void setUp() throws Exception
    {
```



```java
        // rien
    }

    /**
     * Nettoyage après chaque test
     *
     * @throws java.lang.Exception
     */
    @After
    public void tearDown() throws Exception
    {
        point = null;
        System.gc();
    }

    /**
     * Assertion de la valeur de x du point "point"
     *
     * @param message le message associé à l'assertion
     * @param value la valeur attendue
     * @param tolerance la tolérance de la valeur
     */
    private void assertX(String message, double value, double tolerance)
    {
        assertEquals(message, value, point.getX(), tolerance);
    }

    /**
     * Assertion de la valeur de y du point "point"
     *
     * @param message le message associé à l'assertion
     * @param value la valeur attendue
     * @param tolerance la tolérance de la valeur
     */
    private void assertY(String message, double value, double tolerance)
    {
        assertEquals(message, value, point.getY(), tolerance);
    }

    /**
     * Génère un nombre aléatoire compris entre [0...maxValue[
     *
     * @param maxValue la valeur max du nombre aléatoire
     * @return un nombre aléatoire compris entre [0...maxValue[
     */
    private double randomNumber(double maxValue)
    {
        return Math.random() * maxValue;
    }

    /**
     * Génère un nombre aléatoire compris entre [-range...range[
     *
     * @param range l'étendue du nombre aléatoire généré
     * @return un nombre aléatoire compris entre [-range...range[
     */
    private double randomRange(double range)
    {
        return (Math.random() - 0.5) * 2.0 * range;
    }

    /**
     * Test method for {@link points.Point2D#Point2D()}.
     */
    @Test
    public void testPoint2D()
    {
        String testName = new String("Point2D()");
        System.out.println(testName);

        point = new Point2D();

        assertNotNull(testName + " instance", point);
        assertX(testName + ".getX() == 0.0", 0.0, 0.0);
        assertY(testName + ".getY() == 0.0", 0.0, 0.0);
        assertTrue(testName + ".getNbPoints()", Point2D.getNbPoints() > 0);
    }

    /**
     * Test method for {@link points.Point2D#Point2D(double, double)}.
     */
    @Test
    public void testPoint2DDoubleDouble()
    {
        String testName = new String("Point2D(double, double)");
        System.out.println(testName);

        double valueX = 1.0;
        double valueY = Double.NaN;
        point = new Point2D(valueX, valueY);
```

```
181         assertNotNull(testName + " instance", point);
182         assertX(testName + ".getX() == 1.0", valueX, 0.0);
183         assertY(testName + ".getY() == NaN", valueY, 0.0);
184     }
185
186     /**
187      * Test method for {@link points.Point2D#Point2D(points.Point2D)}.
188      */
189     @Test
190     public void testPoint2DPoint2D()
191     {
192         String testName = new String("Point2D(Point2D)");
193         System.out.println(testName);
194
195         Point2D specimen = new Point2D(randomNumber(maxRandom),
196                 randomNumber(maxRandom));
197         assertNotNull(testName + " instance specimen", specimen);
198
199         point = new Point2D(specimen);
200         assertNotNull(testName + " instance copie", point);
201         assertX(testName + ".getX() == " + specimen.getX(), specimen.getX(),
202                 0.0);
203         assertY(testName + ".getY() == " + specimen.getY(), specimen.getY(),
204                 0.0);
205     }
206
207     /**
208      * Test method for {@link points.Point2D#getX()}.
209      */
210     @Test
211     public void testGetX()
212     {
213         String testName = new String("Point2D.getX()");
214         System.out.println(testName);
215
216         point = new Point2D(1.0, 0.0);
217         assertNotNull(testName + "instance", point);
218         assertEquals(testName + ".getX() == 1.0", 1.0, point.getX(), 0.0);
219     }
220
221     /**
222      * Test method for {@link points.Point2D#getY()}.
223      */
224     @Test
225     public void testGetY()
226     {
227         String testName = new String("Point2D.getY()");
228         System.out.println(testName);
229
230         point = new Point2D(0.0, 1.0);
231         assertNotNull(testName + "instance", point);
232         assertEquals(testName + ".getY() == 1.0", 1.0, point.getY(), 0.0);
233     }
234
235     /**
236      * Test method for {@link points.Point2D#setX(double)}.
237      */
238     @Test
239     public void testSetX()
240     {
241         String testName = new String("Point2D.setX(double)");
242         System.out.println(testName);
243
244         point = new Point2D();
245         assertNotNull(testName + " instance", point);
246         assertEquals(testName + ".getX() == 0.0", 0.0, point.getX(), 0.0);
247         point.setX(2.0);
248         assertEquals(testName + ".getX() == 2.0", 2.0, point.getX(), 0.0);
249     }
250
251     /**
252      * Test method for {@link points.Point2D#setY(double)}.
253      */
254     @Test
255     public void testSetY()
256     {
257         String testName = new String("Point2D.setY(double)");
258         System.out.println(testName);
259
260         point = new Point2D();
261         assertNotNull(testName + " instance", point);
262         assertEquals(testName + ".getY() == 0.0", 0.0, point.getY(), 0.0);
263         point.setY(2.0);
264         assertEquals(testName + ".getY() == 2.0", 2.0, point.getY(), 0.0);
265     }
266
267     /**
268      * Test method for {@link points.Point2D#getEpsilon()}.
269      */
270     @Test
```



```
271     public void testGetEpsilon()
272     {
273         String testName = new String("Point2D.getEpsilon()");
274         System.out.println(testName);
275
276         double result = Point2D.getEpsilon();
277         assertEquals(testName, 1e-6, result, 0.0);
278     }
279
280     /**
281      * Test method for {@link points.Point2D#getNbPoints()}.
282      */
283     @Test
284     public void testGetNbPoints()
285     {
286         String testName = new String("Point2D.getNbPoints()");
287         System.out.println(testName);
288
289         point = new Point2D();
290         /*
291          * On ne sait pas combien de points sont encore en mémoire : cela dépend
292          * du Garbage Collector. On peut donc juste vérifier qu'il y en a au
293          * moins un
294          */
295         assertTrue(testName, Point2D.getNbPoints() ≥ 1);
296     }
297
298     /**
299      * Test method for {@link points.Point2D#toString()}.
300      */
301     @Test
302     public void testToString()
303     {
304         String testName = new String("Point2D.toString()");
305         System.out.println(testName);
306
307         point = new Point2D(Math.PI, Math.E);
308         String expectedString = new String(
309                 "x = 3.141592653589793 y = 2.718281828459045");
310         String result = point.toString();
311         assertEquals(testName, expectedString, result);
312     }
313
314     /**
315      * Test method for {@link points.Point2D#deplace(double, double)}.
316      */
317     @Test
318     public void testDeplace()
319     {
320         String testName = new String("Point2D.deplace(double, double)");
321         System.out.println(testName);
322
323         point = new Point2D();
324         double origineX = point.getX();
325         double origineY = point.getY();
326         double deltaX = 5.0;
327         double deltaY = 3.0;
328
329         point.deplace(deltaX, deltaY);
330         assertEquals(testName + ".getX() après +delta", origineX + deltaX,
331                 point.getX(), 0.0);
332         assertEquals(testName + ".getY() après +delta", origineY + deltaY,
333                 point.getY(), 0.0);
334
335         Point2D retour = point.deplace(-deltaX, -deltaY);
336         double tolerance = Point2D.getEpsilon();
337         assertSame(testName + " return == point déplacé", point, retour);
338         assertEquals(testName + ".getX() après –delta", origineX, point.getX(),
339                 tolerance);
340         assertEquals(testName + ".getY() après –delta", origineY, point.getY(),
341                 tolerance);
342     }
343
344     /**
345      * Test method for
346      * {@link points.Point2D#distance(points.Point2D, points.Point2D)}.
347      */
348     @Test
349     public void testDistancePoint2DPoint2D()
350     {
351         String testName = new String("Point2D.distance(Point2D, Point2D)");
352         System.out.println(testName);
353
354         double radius = randomNumber(maxRandom);
355         double angleStep = Math.PI / nbSteps;
356
357         // Distances entre deux points diamétralement opposés le long d'un
358         // cercle
359         for (double angle = 0.0; angle < (Math.PI * 2.0); angle += angleStep)
360         {
```

```
361             points.clear();
362             double x = radius * Math.cos(angle);
363             double y = radius * Math.sin(angle);
364             points.add(new Point2D(x, y));
365             points.add(new Point2D(-x, -y));
366
367             assertEquals(testName + "p0p1[" + String.valueOf(angle) + "]",
368                     radius * 2.0,
369                     Point2D.distance(points.get(0), points.get(1)),
370                     Point2D.getEpsilon());
371             assertEquals(testName + "p1p0[" + String.valueOf(angle) + "]",
372                     radius * 2.0,
373                     Point2D.distance(points.get(1), points.get(0)),
374                     Point2D.getEpsilon());
375         }
376     }
377
378     /**
379      * Test method for {@link points.Point2D#distance(points.Point2D)}.
380      */
381     @Test
382     public void testDistancePoint2D()
383     {
384         String testName = new String("Point2D.distance(Point2D)");
385         System.out.println(testName);
386
387         double origineX = randomRange(maxRandom);
388         double origineY = randomRange(maxRandom);
389         point = new Point2D(origineX, origineY);
390         double radius = randomNumber(maxRandom);
391         double angleStep = Math.PI / nbSteps;
392
393         // Distance entre un point fixe (point) et des points le long
394         // d'un cercle l'entourant (p)
395         for (double angle = 0.0; angle < (Math.PI * 2.0); angle += angleStep)
396         {
397             Point2D p = new Point2D(origineX + (radius * Math.cos(angle)),
398                     origineY + (radius * Math.sin(angle)));
399
400             assertEquals(testName + "this,p[" + String.valueOf(angle) + "]",
401                     radius, point.distance(p), Point2D.getEpsilon());
402             assertEquals(testName + "p,this[" + String.valueOf(angle) + "]",
403                     radius, p.distance(point), Point2D.getEpsilon());
404         }
405     }
406
407     /**
408      * Test method for {@link points.Point2D#equals(java.lang.Object)}.
409      */
410     @Test
411     public void testEqualsObject()
412     {
413         String testName = new String("Point2D.equals(Object)");
414         System.out.println(testName);
415
416         point = new Point2D(randomRange(maxRandom), randomRange(maxRandom));
417         Object o = new Object();
418
419         // Inégalité avec un objet null
420         assertFalse(testName + " sur null", point.equals(null));
421
422         // Inégalité avec un objet de nature différente
423         assertFalse(testName + " sur Object", point.equals(o));
424
425         // Egalité avec soi même
426         Object opoint = point;
427         assertEquals(testName + " sur this", point, opoint);
428
429         // Egalité avec une copie de soi même
430         Point2D otherPoint = new Point2D(point);
431         Object op = otherPoint;
432         assertEquals(testName + " sur copie", point, op);
433         double epsilon = Point2D.getEpsilon();
434
435         // Egalité avec un point déplacé de epsilon au plus
436         for (long i = 0; i < nbTrials; i++)
437         {
438             otherPoint.setX(point.getX());
439             otherPoint.setY(point.getY());
440             double radius = randomNumber(epsilon);
441             double angle = randomNumber(Math.PI * 2.0);
442             otherPoint.deplace(
443                     radius * Math.cos(angle),
444                     radius * Math.sin(angle));
445             double distance = point.distance(otherPoint);
446
447             /*
448              * Attention, à cause des approximations dues aux cos et sin
449              * le déplacement peut être légèrement supérieure à epsilon
450              */
```



```
451             if (distance < epsilon)
452             {
453                 assertEquals(testName + " point déplacé < epsilon [" + distance
454                         + "]", point, otherPoint);
455             }
456             else
457             {
458                 assertFalse(testName + " point déplacé >= epsilon [" + distance
459                         + "]", point.equals(op));
460             }
461         }
462
463         // Inégalité avec un point déplacé
464         for (long i = 0; i < nbTrials; i++)
465         {
466             otherPoint.setX(point.getX());
467             otherPoint.setY(point.getY());
468             otherPoint.deplace(randomRange(maxRandom), randomRange(maxRandom));
469             double distance = point.distance(otherPoint);
470
471             if (distance < epsilon)
472             {
473                 assertEquals(testName + " point déplacé proche [" + distance
474                         + "]", point, otherPoint);
475             }
476             else
477             {
478                 assertFalse(testName + " point déplacé loin [" + distance + "]",
479                         point.equals(otherPoint));
480             }
481         }
482     }
483 }
```

```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.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.HashMap;
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 figures.Cercle;
import figures.Figure;
import figures.Groupe;
import figures.Polygone;
import figures.Rectangle;
import figures.Triangle;
import points.Point2D;

/**
 * Classe de test de l'ensemble des figures
 * @author davidroussel
 */
@RunWith(value = Parameterized.class)
public class FigureTest<F extends Figure>
{
    /**
     * La figure courante à tester
     */
    private F testFigure = null;

    /**
     * La classe de la figure à tester (pour invoquer ses constructeurs)
     */
    private Class<F> figureDefinition = null;

    /**
     * Le nom/type de la figure courante à tester
     */
    private String typeName;

    /**
     * Tolérance pour les comparaisons numériques (aires, distances)
     */
    private static final double tolerance = Point2D.getEpsilon();

    /**
     * Les différentes natures de figures à tester
     */
    @SuppressWarnings("unchecked")
    private static final Class<? extends Figure>[] figureTypes =
    (Class<? extends Figure>[]) new Class<?>[]
    {
        Cercle.class,
        Rectangle.class,
        Triangle.class,
        Polygone.class,
        Groupe.class
    };

    /**
     * L'ensemble des figures à tester
     */
    private static final Figure[] figures = new Figure[figureTypes.length];

    /**
     * Autre ensemble (distinct) de figures à tester pour l'égalité;
     */
    private static final Figure[] altFigures = new Figure[figureTypes.length];

    /**
     * la map permettant d'obtenir la figure en fonction de son nom.
     * Sera construite à partir de {@link #noms} et de {@link #figures}
     */
    private static Map<String, Figure> figuresMap =
            new HashMap<String, Figure>();
```



```java

    /**
     * Les points à utiliser pour construire les figures
     */
    private static final Point2D[][] points = new Point2D[][] {
        {new Point2D(7,3)}, // Cercle
        {new Point2D(4,1), new Point2D(8,4)}, // Rectangle
        {new Point2D(3,2), new Point2D(7,3), new Point2D(4,6)}, // Triangle
        {new Point2D(5,1), new Point2D(8,2), new Point2D(7,5), new Point2D(2,4),
            new Point2D(2,3)} // Polygone
    };

    /**
     * Nom des différentes figures à tester
     */
    private static final String[] noms = new String[figureTypes.length];

    /**
     * Index dans le tableau de noms {@link #noms} à partir d'un nom.
     */
    private static Map<String, Integer> nomsIndex = new HashMap<String, Integer>();


    /**
     * Les différents centre des figures
     */
    private static final Point2D[] centres = new Point2D[] {
        new Point2D(7,3), // Cercle
        new Point2D(6, 2.5), // Rectangle
        new Point2D(4.666666666666667, 3.6666666666666665), // Triangle
        new Point2D(5.150537634408602, 3.053763440860215), // Polygone
        new Point2D() // Groupe : on le calculera plus tard
    };

    /**
     * toString attendu des différentes figures
     */
    private static String[] toStrings = new String[] {
        "Cercle : x = 7.0 y = 3.0, r = 2.0",
        "Rectangle : x = 4.0 y = 1.0, x = 8.0 y = 4.0",
        "Triangle : x = 3.0 y = 2.0, x = 7.0 y = 3.0, x = 4.0 y = 6.0",
        "Polygone : x = 5.0 y = 1.0, x = 8.0 y = 2.0, x = 7.0 y = 5.0, x = 2.0 y = 4.0, x = 2.0 y = 3.0",
        "" // Groupe = à recalculer d'après les précédents
    };

    /**
     * aires attendues des différentes figures
     */
    private static double[] aires = new double[] {
        12.566371, // Cercle
        12.0, // Rectangle
        7.5, // Triangle
        15.5, // Polygone
        47.566371 // Groupe
    };

    /**
     * Distances entre les centres des figures
     */
    private static double[][] interDistances = new double[][] {
    //   Cercle    Rect.     Tri.      Poly.     Grp.
        {0.0,      1.118034, 2.426703, 1.850244, 1.296870}, // Cercle
        {1.118034, 0.0,      1.771691, 1.014022, 0.628953}, // Rectangle
        {2.426703, 1.771691, 0.0,      0.780885, 1.204446}, // Triangle
        {1.850244, 1.014022, 0.780885,      0.0, 0.553765}, // Polygone
        {1.296870, 0.628953, 1.204446, 0.553765,      0.0}  // Groupe
    };

//  /**
//   * Enum interne décrivant les indices des différentes figures dans les
//   * tableaux #figureTypes, #points, #centres, #toStrings, #aires,
//   * #interDistances
//   * @author davidroussel
//   *
//   */
//  static private enum Indices
//  {
//      CIRCLE,
//      RECTANGLE,
//      TRIANGLE,
//      POLYGON,
//      GROUP;
//
//      public int toInt() throws IllegalArgumentException
//      {
//          switch(this)
//          {
//              case CIRCLE:
//                  return 0;
//              case RECTANGLE:
```

```
181 //                  return 1;
182 //              case TRIANGLE:
183 //                  return 2;
184 //              case POLYGON:
185 //                  return 3;
186 //              case GROUP:
187 //                  return 4;
188 //              default :
189 //                  throw new IllegalArgumentException("Indices::toInt");
190 //          }
191 //      }
192 //
193 //      @Override
194 //      public String toString() throws IllegalArgumentException
195 //      {
196 //          switch(this)
197 //          {
198 //              case CIRCLE:
199 //                  return new String("Cercle");
200 //              case RECTANGLE:
201 //                  return new String("Rectangle");
202 //              case TRIANGLE:
203 //                  return new String("Triangle");
204 //              case POLYGON:
205 //                  return new String("Polygone");
206 //              case GROUP:
207 //                  return new String("Groupe");
208 //              default :
209 //                  throw new IllegalArgumentException("Indices::toString");
210 //          }
211 //
212 //      }
213 //  }
214
215     /**
216      * la map permettant d'obtenir le centre précalculé d'une figure en fonction
217      * de son nom.
218      * Sera construite à partir de {@link #noms} et de {@link #centres}
219      */
220     private static Map<String, Point2D> centresMap =
221             new HashMap<String, Point2D>();
222
223     /**
224      * Un point à l'intérieur de toutes les figures
225      */
226     private static final Point2D insidePoint = new Point2D(6,3);
227
228     /**
229      * Un point à l'extérieur de toutes les figures
230      */
231     private static final Point2D outsidePoint = new Point2D(6,5);
232
233     /**
234      * Mise en place avant l'ensemble des tests
235      * @throws java.lang.Exception
236      */
237     @BeforeClass
238     public static void setUpBeforeClass() throws Exception
239     {
240         // remplissage des noms
241         for (int i = 0; i < figureTypes.length; i++)
242         {
243             noms[i] = figureTypes[i].getSimpleName();
244         }
245
246         /*
247          * Premier ensmble de figures
248          */
249         // Première figure = cercle
250         figures[0] = new Cercle(points[0][0], 2);
251         // Seconde figure = rectangle
252         figures[1] = new Rectangle(points[1][0], points[1][1]);
253         // Troisième figure = triangle
254         figures[2] = new Triangle(points[2][0], points[2][1], points[2][2]);
255         // Quatrième figure = polygone
256         ArrayList<Point2D> polyPoints = new ArrayList<Point2D>();
257         for (Point2D p : points[3])
258         {
259             polyPoints.add(p);
260         }
261         figures[3] = new Polygone(polyPoints);
262         // Cinquième figure : groupe de l'ensemble des 4 premières
263         ArrayList<Figure> figureGroup = new ArrayList<Figure>();
264         for (int i = 0; i < (figures.length - 1); i++)
265         {
266             figureGroup.add(figures[i]);
267         }
268         figures[4] = new Groupe(figureGroup);
269
270         /*
```



```
271          * Second ensemble de figures
272          */
273         // Première figure = cercle
274         altFigures[0] = new Cercle(points[0][0], 2);
275         // Seconde figure = rectangle
276         altFigures[1] = new Rectangle(points[1][1], points[1][0]);
277         // Troisième figure = triangle
278         altFigures[2] = new Triangle(points[2][1], points[2][0], points[2][2]);
279         // Quatrième figure = polygone
280         polyPoints.clear();
281         for (int i = 1; i ≤ points[3].length; i++)
282         {
283             polyPoints.add(points[3][i%points[3].length]);
284         }
285         altFigures[3] = new Polygone(polyPoints);
286         // Cinquième figure : groupe de l'ensemble des 4 premières
287         figureGroup.clear();
288         for (int i = figures.length - 2; i ≥ 0; i--)
289         {
290             figureGroup.add(altFigures[i]);
291         }
292         altFigures[4] = new Groupe(figureGroup);
293
294         // calcul du barycentre des 4 premières figures pour initialiser
295         // le centre du groupe de figures
296         int j = 0;
297         double centreX = 0.0;
298         double centreY = 0.0;
299         for (; j < (figures.length - 1); j++)
300         {
301             Point2D centre = figures[j].getCentre();
302             centreX += centre.getX();
303             centreY += centre.getY();
304         }
305
306         centres[4].setX(centreX / j);
307         centres[4].setY(centreY / j);
308
309         // calcul du toString des Groupes
310         StringBuilder sb = new StringBuilder("Groupe : ");
311         for (int i = 0; i < (toStrings.length - 1); i++)
312         {
313             sb.append("\n" + toStrings[i]);
314         }
315         toStrings[4] = sb.toString();
316
317         // construction des maps de
318         //  - figures
319         //  - centres
320         for (int i = 0; i < figureTypes.length; i++)
321         {
322             figuresMap.put(noms[i], figures[i]);
323             centresMap.put(noms[i], centres[i]);
324             nomsIndex.put(noms[i], Integer.valueOf(i));
325         }
326     }
327
328     /**
329      * Nettoyage après l'ensemble des tests
330      * @throws java.lang.Exception
331      */
332     @AfterClass
333     public static void tearDownAfterClass() throws Exception
334     {
335         // rien
336     }
337
338     /**
339      * Mise en place avant chaque test
340      * @throws java.lang.Exception
341      */
342     @Before
343     public void setUp() throws Exception
344     {
345         // rien
346     }
347
348     /**
349      * Nettoyage après chaque test
350      * @throws java.lang.Exception
351      */
352     @After
353     public void tearDown() throws Exception
354     {
355         // rien
356     }
357
358     /**
359      * Paramètres à transmettre au constructeur de la classe de test.
360      *
```

```
361      * @return une collection de tableaux d'objet contenant les paramètres à
362      *         transmettre au constructeur de la classe de test
363      */
364     @Parameters(name = "{index}:{1}")
365     public static Collection<Object[]> data()
366     {
367         Object[][] data = new Object[figureTypes.length][2];
368         for (int i = 0; i < figureTypes.length; i++)
369         {
370             data[i][0] = figureTypes[i];
371             data[i][1] = figureTypes[i].getSimpleName();
372         }
373         return Arrays.asList(data);
374     }
375
376     /**
377      * Constructeur parametré par le type de figure à tester
378      * @param typeFigure le type de figure à tester
379      * @param typeName le nom du type à tester (pour affichage)
380      */
381     @SuppressWarnings("unchecked") // à cause du cast en F
382     public FigureTest(Class<F> typeFigure, String typeName)
383     {
384         figureDefinition = typeFigure;
385         this.typeName = typeName;
386         testFigure = (F) figuresMap.get(typeName);
387     }
388
389     /**
390      * Test method for one of {@link figures.Figure} default constructor
391      */
392     @Test
393     public final void testFigureConstructor()
394     {
395         String testName = new String(typeName + "()");
396         System.out.println(testName);
397         Constructor<F> defaultConstructor = null;
398         Class<?>[] constructorsArgs = new Class<?>[0];
399
400         try
401         {
402             defaultConstructor =
403                 figureDefinition.getConstructor(constructorsArgs);
404         }
405         catch (SecurityException e)
406         {
407             fail(testName +  " constructor security exception");
408         }
409         catch (NoSuchMethodException e)
410         {
411             fail(testName +  " constructor not found");
412         }
413
414         if (defaultConstructor ≠ null)
415         {
416             Object instance = null;
417             try
418             {
419                 instance = defaultConstructor.newInstance(new Object[0]);
420             }
421             catch (IllegalArgumentException e)
422             {
423                 fail(testName + " wrong constructor arguments");
424             }
425             catch (InstantiationException e)
426             {
427                 fail(testName + " instanciation exception");
428             }
429             catch (IllegalAccessException e)
430             {
431                 fail(testName + " illegal access");
432             }
433             catch (InvocationTargetException e)
434             {
435                 fail(testName + " invocation target exception");
436             }
437
438             assertNotNull(testName, instance);
439             assertEquals(testName + " self equality", instance, instance);
440         }
441     }
442
443     /**
444      * Test method for one of {@link figures.Figure} copy constructor
445      */
446     @Test
447     public final void testFigureConstructorFigure()
448     {
449         String testName = new String(typeName + "(" + typeName + ")");
450         System.out.println(testName);
```



```
451         Constructor<F> copyConstructor = null;
452         Class<?>[] constructorsArgs = new Class<?>[] {figureDefinition};
453
454         try
455         {
456             copyConstructor =
457                 figureDefinition.getConstructor(constructorsArgs);
458         }
459         catch (SecurityException e)
460         {
461             fail(testName +  " constructor security exception");
462         }
463         catch (NoSuchMethodException e)
464         {
465             fail(testName +  " constructor not found");
466         }
467
468         if (copyConstructor ≠ null)
469         {
470             Object instance = null;
471             try
472             {
473                 instance = copyConstructor.newInstance(testFigure);
474             }
475             catch (IllegalArgumentException e)
476             {
477                 fail(testName + " wrong constructor arguments");
478             }
479             catch (InstantiationException e)
480             {
481                 fail(testName + " instanciation exception");
482             }
483             catch (IllegalAccessException e)
484             {
485                 fail(testName + " illegal access");
486             }
487             catch (InvocationTargetException e)
488             {
489                 fail(testName + " invocation target exception");
490             }
491
492             assertNotNull(testName, instance);
493             assertEquals(testName + " equality", testFigure, instance);
494         }
495     }
496
497     /**
498      * Test method for {@link figures.Figure#getNom()}.
499      */
500     @Test
501     public final void testGetNom()
502     {
503         String testName = new String(typeName + ".getNom()");
504         System.out.println(testName);
505
506         assertEquals(testName, noms[nomsIndex.get(typeName).intValue()],
507                 testFigure.getNom());
508     }
509
510     /**
511      * Test method for {@link figures.Figure#deplace(double, double)}.
512      */
513     @Test
514     public final void testDeplace()
515     {
516         String testName = new String(typeName + ".deplace(double, double)");
517         System.out.println(testName);
518
519         Point2D centreBefore = new Point2D(testFigure.getCentre());
520
521         double dx = 1.0;
522         double dy = 1.0;
523
524         testFigure.deplace(dx, dy);
525
526         Point2D centreAfter = testFigure.getCentre();
527
528         assertEquals(testName, centreBefore.deplace(dx, dy), centreAfter);
529
530         testFigure.deplace(-dx, -dy);
531     }
532
533     /**
534      * Test method for {@link figures.Figure#toString()}.
535      */
536     @Test
537     public final void testToString()
538     {
539         String testName = new String(typeName + ".toString()");
540         System.out.println(testName);
```

```
541
542         assertEquals(testName, toStrings[nomsIndex.get(typeName).intValue()],
543                 testFigure.toString());
544     }
545
546     /**
547      * Test method for {@link figures.Figure#contient(points.Point2D)}.
548      */
549     @Test
550     public final void testContient()
551     {
552         String testName = new String(typeName + ".contient(Point2D)");
553         System.out.println(testName);
554
555         assertTrue(testName + " inner point", testFigure.contient(insidePoint));
556
557         assertFalse(testName + " outer point", testFigure.contient(outsidePoint));
558     }
559
560     /**
561      * Test method for {@link figures.Figure#getCentre()}.
562      */
563     @Test
564     public final void testGetCentre()
565     {
566         String testName = new String(typeName + ".getCentre()");
567         System.out.println(testName);
568
569         assertEquals(testName, centres[nomsIndex.get(typeName).intValue()],
570                 testFigure.getCentre());
571     }
572
573     /**
574      * Test method for {@link figures.Figure#aire()}.
575      */
576     @Test
577     public final void testAire()
578     {
579         String testName = new String(typeName + ".aire()");
580         System.out.println(testName);
581
582         assertEquals(testName, aires[nomsIndex.get(typeName).intValue()],
583                 testFigure.aire(), tolerance);
584     }
585
586     /**
587      * Test method for {@link figures.Figure#distanceToCentreOf(figures.Figure)}.
588      */
589     @Test
590     public final void testDistanceToCentreOf()
591     {
592         String testName = new String(typeName + ".distanceToCentreOf(Figure)");
593         System.out.println(testName);
594
595         for (int i = 0; i < figures.length; i++)
596         {
597             assertEquals(testName + "->" + noms[i],
598                     interDistances[nomsIndex.get(typeName).intValue()][i],
599                     testFigure.distanceToCentreOf(figures[i]), tolerance);
600         }
601     }
602
603     /**
604      * Test method for {@link figures.Figure#equals(java.lang.Object)}.
605      */
606     @Test
607     public final void testEquals()
608     {
609         String testName = new String(typeName + ".equals(Object)");
610         System.out.println(testName);
611
612         // Inégalité avec null
613         assertFalse(testName + " != null", testFigure.equals(null));
614
615         // Egalité avec soi même
616         assertEquals(testName + " == this", testFigure, testFigure);
617
618         // Egalité / Inégalité avec le même ensemble de figures
619         for (int i = 0; i < figures.length; i++)
620         {
621             if (nomsIndex.get(typeName).intValue() == i)
622             {
623                 assertEquals(testName + " == (1) " + noms[i], testFigure,
624                         figures[i]);
625             }
626             else
627             {
628                 assertFalse(testName + " != (1) " + noms[i],
629                         testFigure.equals(figures[i]));
630             }
```



```
631         }
632
633         // Egalité / Inégalité avec l'autre ensemble de figures
634         for (int i = 0; i < figures.length; i++)
635         {
636             if (nomsIndex.get(typeName).intValue() == i)
637             {
638                 assertEquals(testName + " == (2) " + noms[i], testFigure,
639                         altFigures[i]);
640             }
641             else
642             {
643                 assertFalse(testName + " != (2) " + noms[i],
644                         testFigure.equals(altFigures[i]));
645             }
646         }
647     }
648 }
```