TD 1: Classes et objets
Correction

Exercice 1. Trace d'un programme

Que fournit le programme suivant ?

```
class Entier
{
        public Entier (int nn) {n = nn;}
        public void incr (int dn) {n += dn;}
        public void imprime () {System.out.println (n);}
        private int n ;
}

public class Ex1
{
        public static void main (String args[])
        {
                Entier n1 = new Entier (2);
                System.out.print ("n1 = "); n1.imprime();
                Entier n2 = new Entier (5);
                System.out.print ("n2 = "); n2.imprime() ;
                n1.incr(3) ;
                System.out.print ("n1 = "); n1.imprime() ;
                System.out.println ("n1 == n2 est " + (n1 == n2));
                n1 = n2; n2.incr(12);
                System.out.print ("n2 = "); n2.imprime() ;
                System.out.print ("n1 = "); n1.imprime() ;
                System.out.println ("n1 == n2 est " + (n1 == n2)) ;
        }
}
```

Correction

```
n1 = 2
n2 = 5
n1 = 5
n1 == n2 est false
n2 = 17
n1 = 17
n1 == n2 est true
```

L'égalité n1 == n2 teste si n1 et n2 font référence au même objet en mémoire. Le premier test d'égalité est faux car n1 et n2 ne font pas référence au même objet (chacun a été initialisé avec son propre **new**). Le second test est vrai car il fait suite à l'affectation n1 = n2, après lequel n1 fait maintenant référence au même objet que n2.

Exercice 2. Trace d'un programme

a. Détailler les différentes valeurs présentes dans les champs de la classe A lors de l'exécution du programme suivant. Quel est le résultat de l'exécution ?

```
class A
{
        public A (int coeff)
        {
                nbre *= coeff;
                nbre += decal;
        }
```

```
        public void affiche ()
        {
                System.out.println ("nbre = " + nbre + " decal = " + decal);
        }
        private int nbre = 20 ;
        private int decal;
}

public class Ex2
{
        public static void main (String args[])
        {
                A a = new A (5);
                a.affiche();
        }
}
```

Correction
Rappelons les trois étapes d'initialisation des attributs en Java :

- **Initialisation par défaut** : les attributs sont initialisés avec leur valeur par défaut (qui dépend de leur type) :

Type du champ	Valeur par défaut
boolean	false
char	caractère de code nul
byte, int, short, long	0
flottant	0.f ou 0
Objet	null

- **Initialisation explicite** : les attributs prennent la valeur d'initialisation explicite.
- **Exécution du constructeur** : le code du constructeur appelé est exécuté.

Dans le programme donné, les variables prennent les valeurs suivantes successivement.

	Initialisation par défaut	Initialisation explicite	Exécution du constructeur
nbre	0	20	100
decal	0	0	0

L'exécution affiche la chaîne "nbre = 100 decal = 0"

b. Comment faut-il modifier l'initialisation par des champs de la classe A pour que la même exécution du programme affiche *nbre = 103...* ?

Correction
Il faut modifier la déclaration de decal par **private int** decal = 3;

Exercice 3. Géométrie plane

Définir des classes représentant

1. les points de $\mathbb{R}^2$ (on utilisera le type flottant `double` pour représenter une approximation d'un nombre réel). Cette classe comprendra aussi une méthode calculant la distance à un autre point.
2. les droites du plan (représentées par un couple de points). Elle disposera en outre des méthodes suivantes :
   - test d'appartenance
   - test de parallélisme
   - test d'orthogonalité
   - une fonction retournant la droite parallèle à la droite courante passant par un point donné
3. les triangles du plan. La classe disposera en outre des méthodes suivantes :
   - test si le triangle est isocèle
   - test si le triangle est rectangle
   - calcul de chacune des trois hauteurs, et médianes
   - calcul du centre de gravité

Correction

```java
/**
 * Classe de points en deux dimensions.
 */
public class Point {

	/** Constructeur */
	public Point(){}

	/** Constructeur à partir de doubles */
	public Point(double x, double y) {
		this.x = x;
		this.y = y;
	}

	/** Constructeur par copie */
	public Point(Point p) {
		this.x = p.x;
		this.y = p.y;
	}

	/**
	 * Permute le point donné avec le point courant.
	 * @param	a	le point avec lequel permuter
	 */
	public void permute (Point a) {
		Point c = new Point (0,0); // création d'un point intermédiaire
		c.x = a.x; c.y = a.y; // copie de a dans c
		a.x = x; a.y = y; // copie du point courant dans a
		x = c.x; y = c.y; // copie de c dans le point courant
	}

	/** @return l'abscisse du point */
	public double getx() {return x;}

	/** @return l'ordonnée du point */
	public double gety() {return y;}

	/**
	 * Modifie l'abscisse du point
	 * @param	x	l'abscisse du point
	 */
	public void setx(double x) {this.x = x;}

	/**
	 * Modifie l'ordonnée du point
	 * @param	y	l'ordonnée du point
	 */
	public void sety(double y) {this.y = y;}

	/** Affiche les coordonnées du point. */
	public void affiche() {
		System.out.println("Point x = " + x + "; y = " + y);
	}

	/** @return une chaîne représentant le point */
	public String toString() {
		return ("Point x = " + x + "; y = " + y);
	}

	/**
	 * Calcule la distance entre le point courant et le point donné.
	 * @param	p	le point
	 * @return		la distance avec le point p
	 */
	public double distance(Point p) {
		double xdP = x - p.getx();
		double ydP = y - p.gety();
		return Math.sqrt(Math.pow(xdP,2)+Math.pow(ydP,2));
	}

	/**
	 * Addition d'un point donné au point courant.
```

```
         * @param         p         le point à ajouter.
         */
        public void ajouter(Point p) {
                x += p.x;
                y += p.y;
        }

        /** Abscisse du point. */
        private double x;
        /** Ordonnée du point. */
        private double y;
}

/**
 * Classe de droites en deux dimensions.
 * Une droite est représentée par deux points.
 */
public class Droite {
        /** Constructeur. */
        public Droite(){}

        /** Constructeur à partir de points. */
        public Droite(Point a, Point b) {
                this.a = new Point(a);
                this.b = new Point(b);
        }

        /** Affiche la droite. */
        public void affiche() {
                System.out.println("Droite de " + a + "; " + b);
        }

        /** @return le premier point définissant la droite */
        public Point geta() {return a;}

        /** @return le second point définissant la droite */
        public Point getb() {return b;}

        /**
         * Modifie le premier point de la droite
         * @param         a         le point
         */
        public void seta(Point a) {this.a = a;}

        /**
         * Modifie le second point de la droite
         * @param         b         le point
         */
        public void setb(Point b) {this.b = b;}

        /**
         * Teste l'appartenance d'un point à la droite courante.
         * @param         c         le point à tester
         * @return                  true si le point appartient à la droite courante
         */
        public boolean appartenance(Point c) {
                double xA = this.a.getx();
                double xB = this.b.getx();
                double xC = c.getx();
                double yA = this.a.gety();
                double yB = this.b.gety();
                double yC = c.gety();

                return Utils.is_zero((xA-xB)*(yA-yC) - (xA-xC)*(yA-yB));
        }

        /**
         * Teste le parallelisme d'une droite donnée avec la droite courante.
         * @param         d         la droite à tester
         * @return                  true si la droite donnée est parallèle à la droite courante
         */
        public boolean paralelle(Droite d) {
                double xA = this.a.getx();
                double xB = this.b.getx();
                double xC = d.geta().getx();
```

```
                double xD = d.getb().getx();
                //
                double yA = this.a.gety();
                double yB = this.b.gety();
                double yC = d.geta().gety();
                double yD = d.getb().gety();

                return Utils.is_zero((xA-xB)*(yC-yD) - (xC-xD)*(yA-yB));
        }

        /**
         * Teste l'orthogonalité d'une droite.
         * @param        d        la droite à tester
         * @return                 true si la droite donnée est orthogonale à la droite courante
         */
        public boolean perpendiculaire(Droite d) {
                double xA = this.a.getx();
                double xB = this.b.getx();
                double xC = d.geta().getx();
                double xD = d.getb().getx();
                //
                double yA = this.a.gety();
                double yB = this.b.gety();
                double yC = d.geta().gety();
                double yD = d.getb().gety();

                return Utils.is_zero((xA-xB)*(xC-xD)+(yA-yB)*(yC-yD));
        }

        /**
         * Renvoie une droite parallèle à la droite courante, passant par un point donné.
         * @param         p         le point
         * @return                  une droite parallèle passant par p
         */
        public Droite give_para(Point p) {
                double xA = this.a.getx();
                double xB = this.b.getx();
                double xC = p.getx();
                double xD = xC + (xA-xB);
                //
                double yA = this.a.gety();
                double yB = this.b.gety();
                double yC = p.gety();
                double yD = yC + (yA-yB);

                return new Droite(p, new Point(xD,yD));
        }

        /** Premier point définissant la droite. */
        private Point a;

        /** Second point définissant la droite. */
        private Point b;
}

/**
 * Classe de triangles.
 */
public class Triangle {
        /** Constructeur à partir de trois points. */
        public Triangle(Point a, Point b, Point c) {
                this.a = new Point(a);
                this.b = new Point(b);
                this.c = new Point(c);
        }

        /** Affiche le triangle. */
        public void affiche() {
                System.out.println("Triangle de " + a + "; " + b + "; " + c);
        }

        /** @return le premier point du triangle */
        public Point geta() {return a;}

        /** @return le second point du triangle */
```

```
        public Point getb() {return b;}

        /** @return le troisième point du triangle */
        public Point getc() {return c;}

        /**
         * Modifie le premier point du triangle
         * @param         a         le point
         */
        public void seta(Point a) {this.a = a;}

        /**
         * Modifie le second point du triangle
         * @param         b         le point
         */
        public void setb(Point b) {this.b = b;}

        /**
         * Modifie le troisième point du triangle
         * @param         c         le point
         */
        public void setc(Point c) {this.c = c;}

        /** @return               true si le triangle est isocèle */
        public boolean est_iso() {
                double ab = a.distance(b);
                double ac = a.distance(c);
                double bc = b.distance(c);

                return Utils.is_zero(ab - ac) || Utils.is_zero(ab - bc);
        }

        /** @return               true si le triangle est rectangle */
        public boolean est_rectangle() {
                double ab2 = Math.pow(a.distance(b), 2);
                double ac2 = Math.pow(a.distance(c), 2);
                double bc2 = Math.pow(b.distance(c), 2);

                return Utils.is_zero(ab2 + ac2 - bc2)
                        || Utils.is_zero(ac2 + bc2 - ab2)
                        || Utils.is_zero(ab2 + bc2 - ac2);
        }

        /** @return               le centre de gravité du triangle */
        public Point barycentre() {
                Point g = new Point(a);
                g.ajouter(b);
                g.ajouter(c);
                g.setx(g.getx()/3.0);
                g.sety(g.gety()/3.0);
                return g;
        }

        private Point a;
        private Point b;
        private Point c;
}

public class Utils {
        static final double epsilon = Math.pow(10,-14);

        /** @return                true si le double donné est égal à 0
         * (considéré vrai s'il est inférieur en valeur absolue à un epsilon).
         */
        public static boolean is_zero(double x) {
                return Math.abs(x) < epsilon;
        }
}
```

Exercice 4. Conception de classes

On souhaite écrire un logiciel de gestion du trafic ferroviaire. Il faut que ce logiciel représente

Les voies, définies par
- deux gares (marquant chaque extrémité)
- une longueur en km

Les gares, définies par
- un nom (chaîne de caractère)
- un ensemble de voies qui lui sont connectées.

On pourra utiliser les collections ArrayList<Voie> ou HashSet<Voie> pour stocker l'ensemble de voies d'une gare.

a. Écrire ces deux classes Voie et Gare

On souhaite maintenant représenter des lignes, c'est à dire un chemin d'une gare à une autre, passant éventuellement par plusieurs gares intermédiaires en suivant des voies.

b. Écrire une telle classe.

c. Écrire une classe statique Outils contenant une méthode distance. Cette méthode devra pouvoir prendre en argument
- une voie
- une ligne
- deux gares

et retourner la (meilleure) distance correspondante.

Correction

Comme une Voie se définit par deux gares, et qu'une gare possède une liste de voies, il va falloir définir l'un avant l'autre pour éviter une dépendance circulaire. On choisit ici de définir d'abord une Gare par son nom, puis de créer les voies. On peut noter que les voies s'ajoutent directement aux listes de voies des gares (via la méthode addVoie) dans le constructeur de Voie.

```
// Nécessaire pour utiliser ArrayList
import java.util.ArrayList;

/**
 * Classe de Gare, représentée par un nom et une liste de voies.
 */
class Gare {

    /** Constructeur */
    public Gare(String nom) {
        this.nom = nom;
        voies = new ArrayList<Voie>();
    }

    /** Ajoute une voie à la gare. */
    public void addVoie(Voie v) {
        voies.add(v);
    }

    /**
     * @return      la voie qui relie la gare g à la gare courante
     * @return      null si elles ne sont pas reliées par une voie */
    public Voie getVoie(Gare g) {
        for (Voie v : voies) {
            if (v.arrivee == g || v.depart == g) {
                return v;
            }
        }
        return null;
    }

    private String nom;
    /** Liste des voies passant par cette gare. */
    private ArrayList<Voie> voies;
}

/**
 * Classe de voie.
 * On suppose qu'une voie peut-être empruntée dans les deux sens.
 */
class Voie {
```

```
    public Voie(Gare depart, Gare arrivee, int longueur){
        this.depart = depart;
        this.arrivee = arrivee;
        this.longueur = longueur;
        depart.addVoie(this);
        arrivee.addVoie(this);
    }

    Gare depart;
    Gare arrivee;
    /** Longueur de la voie en km. */
    int longueur;
}

/** Classe de ligne, c'est à dire un chemin passant pas plusieurs gares. */
class Ligne {

    /** Constructeur */
    public Ligne() {
        arrets = new ArrayList<Gare>();
        chemin = new ArrayList<Voie>();
    }

    /** Constructeur à partir d'une gare de départ. */
    public Ligne(Gare g) {
        this();
        arrets.add(g);
    }

    /** @return     la dernière gare du chemin
        @return     null si le chemin est vide */
    public Gare terminus() {
        if (arrets.isEmpty()) {
            return null;
        } else {
            return arrets.get(arrets.size() - 1);
        }
    }

    /**
     * Ajoute un arrêt à la ligne.
     * @param   g   La gare à ajouter
     */
    public void addArret(Gare g){
        if (arrets.isEmpty()) { // Si le chemin est vide
            arrets.add(g);
        } else {
            // On vérifie que la gare précédente et la nouvelle sont bien reliées par une voie
            Gare precedente = terminus();
            Voie v = precedente.getVoie(g);
            if (v != null) { // si la voie existe
                arrets.add(g);
                chemin.add(v);
            } else {
                System.out.println("La gare n'est pas reliée à la ligne");
            }
        }
    }

    /**
     * Gares de la ligne.
     * Elles sont stockées dans l'ordre, d'un terminus à l'autre.
     */
    ArrayList<Gare> arrets;

    /**
     * Voies par lesquelles passe la ligne.
     * Elles sont stockées dans l'ordre, d'un terminus à l'autre.
     */
    ArrayList<Voie> chemin;
}

class Utils {

    /**
```

```java
     * Calcule la distance entre deux gares via une ligne donnée.
     * @param   l   la ligne
     * @param   g1  une gare de l
     * @param   g2  une autre gare de l
     * @return      la distance entre les gares g1 et g2 via l
     */
    public static int distance(Ligne l, Gare g1, Gare g2) {
        boolean depart_atteint = false;
        int d = 0;
        for (int i = 0 ; i < l.arrets.size() ; i++) {
            if (depart_atteint) {
                d += l.chemin.get(i-1).longueur;
                if (g1 == l.arrets.get(i) || g2 == l.arrets.get(i)) {
                    // On est arrivé
                    break;
                }
            } else {
                if (g1 == l.arrets.get(i) || g2 == l.arrets.get(i)) {
                    depart_atteint = true;
                }
            }
        }
        return d;
    }

}

public class Ex4 {
    public static void main(String[] args) {
        Gare grenoble = new Gare("Grenoble");
        Gare chambery = new Gare("Chambery");
        Gare lyon = new Gare("Lyon");
        Gare paris = new Gare("Paris");
        Gare valence = new Gare("Valence");

        new Voie(grenoble, chambery, 70);
        new Voie(lyon, grenoble, 100);
        new Voie(lyon, chambery, 120);
        new Voie(lyon, paris, 600);
        new Voie(paris, valence, 550);
        new Voie(grenoble, valence, 90);
        new Voie(lyon, valence, 100);

        Ligne paris_grenoble = new Ligne(paris);
        paris_grenoble.addArret(lyon);
        paris_grenoble.addArret(grenoble);

        Ligne paris_chambery = new Ligne(paris);
        paris_chambery.addArret(lyon);
        paris_chambery.addArret(chambery);

        System.out.println(Utils.distance(paris_grenoble, paris, grenoble));

    }
}
```

Exercice 5. Compter les instances d'une classe

Écrivez une classe Point possédant une méthode nbObjets qui renvoie le nombre d'instances de Point créées.

Correction

```java
class Point {

    Point() {
        n += 1; // Lors de la création d'un point, on incrémente n
    }

    Point(int x, int y) {
```

```
        this(); // Appel au premier constructeur
        this.x = x;
        this.y = y;
    }

    static public int nbObjets() {
        return n;
    }

    /** Cet attribut stocke le nombre d'instances créées. */
    static private int n = 0;
    private int x;
    private int y;
}

class Ex5 {

    public static void main(String[] args) {
        System.out.println("Points créés : " + Point.nbObjets());
        Point p = new Point(0,1);
        // On peut arbitrairement appeler Point.nbObjets()) ou p.nbObjets())
        System.out.println("Points créés : " + p.nbObjets());
        Point p2 = new Point();
        System.out.println("Points créés : " + Point.nbObjets());
    }
}
```

Exercice 6. L'Unique

Écrivez une classe AnneauUnique qui n'autorise la création que d'une seule instance.

Correction
Pour éviter la création d'instances supplémentaires, on rend le constructeur d'AnneauUnique privé. La classe va stocker l'instance en tant qu'attribut statique. L'accès à cette instance se fait grâce à la méthode statique getAnneau.

```
class AnneauUnique {

    /** Constructeur privé. */
    private AnneauUnique(){
        System.out.println("L'anneau Unique est forgé");
    }

    /** Renvoie l'anneau unique. */
    public static AnneauUnique getAnneau(){
        if (instance == null) {
            instance = new AnneauUnique();
        }
        return instance;
    }

    /** Stocke l'unique instance d'AnneauUnique. */
    private static AnneauUnique instance = null;
}

public class Ex6 {

    public static void main(String[] args) {
        // Création de l'anneau unique
        AnneauUnique anneau = AnneauUnique.getAnneau();
        // Tentative de création d'un second anneau unique.
        AnneauUnique anneau2 = AnneauUnique.getAnneau();
        // Les deux anneaux sont en fait le même !
        System.out.println(anneau == anneau2);
    }
}
```