Licence Informatique **UE J1IN4W01**
Université de Bordeaux **Programmation 2**
2015-2016 **Travaux Dirigés - feuille 5**

Héritage de code, factorisation et classes abstraites

Exercice 1. On considère l'interface `Point2D` ci-dessous (étudiée en td).

```
public interface Point2D{
     double getAbscisse();
     void setAbscisse(double x);
     double getOrdonnee();
     void setOrdonnee(double y);
     double getModule();
     void setModule(double m);
     double getArgument();
     void setArgument(double a);
     Point2D clone();
     boolean equals(Point2D o);
     String toString();
     void translation(double dx, double dy);
     void homothetie(double rapport);
     void rotation(double angle);
}
```

ainsi que les classes `Point2DCartesien` et `Point2DPolaire` qui implémentent l'interface `Point2D` (héritage de type). Quelle partie de l'implémentation de ces deux classes pourrait-elle être factorisée ? De quelle manière proposez-vous de faire la factorisation ?

Prenons maintenant l'interface `ShapeStack`

```
public interface ShapeStack {
    boolean isEmpty();
    void pop();
    void push(Shape s);
    Shape top();
    boolean equals(ShapeStack s);
    ShapeStack clone();
}
```

et les classes `ShapeStackExtensibleSizeArray` et `ShapeStackFixedSizeArray`. Factorisez le code de ces classes.

Exercice 2. On considère la classe `Relation` ci-dessous (étudiée en td).

```
public class Relation{

    public Relation(boolean[][] tRelation){
        int n = tRelation.length;
        relation = new boolean[n][n];
        for(int i=0; i<n; ++i)
            for(int j=0; j<n; ++j)
                relation[i][j] = tRelation[i][j];
    }

    public boolean inRelation(int i, int j){
        return relation[i][j];
    }

    public boolean isReflexive(){
```

```
        for(int i=0; i<relation.length; ++i)
            if (!inRelation(i, i))
                return false;
        return true;
    }

    public boolean isSymetrique(){
        for(int i=1; i<relation.length; ++i)
            for(int j=0; j<i; ++j)
                if (inRelation(i, j)!= inRelation(j, i))
                    return false;
        return true;
    }

    public boolean isTransitive(){
        int n = relation.length;
        for(int i=0; i<n; ++i)
            for(int j=0 ; j<n; ++j)
                if (inRelation(i, j))
                    for(int k=0; k<n; ++k)
                        if (inRelation(j, k) && !inRelation(i, k))
                            return false;
        return true;
    }

    public static Relation composed(Relation r1, Relation r2){
        int n = r1.relation.length;
        boolean[][] composed = new boolean[n][n];
        for(int i=0; i<n; ++i)
            for(int j=0 ; j < n ; ++j)
                if (r1.inRelation(i, j))
                    for(int k=0; k < n; ++k)
                        if(r2.inRelation(j, k))
                            composed[i][k]= true;
        return new Relation(composed);
    }

    public int[] image(int i){
        // on determine la taille de l'image
        int taille=0;
        int n = relation.length;
        for( int j=0; j < n; ++j)
            if(inRelation(i, j))
                taille++;
        int tableImage[] = new int[taille];
        for( int j=0,k=0; j < n; ++j)
            if(inRelation(i, j))
                tableImage[k++]=j;
        return tableImage;
    }

    private boolean[][] relation;

}
```

Proposez une nouvelle organisation du code tel que la définition de la relation soit indépendante des méthodes `isReflexive`, `isTransitive`, etc. On veut pouvoir définir différents types de relations binaires, données par une matrice de relation ou par une fonction mathématique (`IsMultipleOf`, `isInferiorTo`).