DISVE **Pôle Licence**	**ANNÉE UNIVERSITAIRE 2008/2009** **SESSION 1 DE PRINTEMPS 2009** **Parcours :** CSB4, MHT43 **UE :** INF252 **Epreuve : Programmation 2** **Date :** 19 Mai 2009 **Heure :** 11H **Durée :** 1h 30 **Documents :** Tous documents interdits **Vous devez répondre sur une copie d'examen comportant tous les renseignements administratifs. Le sujet comporte 4 pages.** **Epreuve de** Monsieur Domenger Jean-Philippe	

La notation tiendra compte de la clarté et de la précision de l'écriture des réponses. Lisez attentivement le sujet certaines questions font appel à des connaissances du cours indépendamment des exercices.

Barème indicatif

- Exercice 1 – Vecteur Extensible d'objets- 4 points.
- Exercice 2 – Algorithme d'exécution des services d'instances - 6 points
- Exercice 3 – Exercice de synthèse. 10 points

Exercice 1.

Soit l'interface VecteurObject.

```
public interface VecteurObject
{
        public void ajouter(int indice, Object élement) ;
        // ajoute élément à la position indice dans le vecteur.
        public void supprimer(int indice) ;
        // met null à la position indice dans le vecteur
        public int longueur() ;
        // retourne la position maximale d'un élément non null présent dans le vecteur.
        public int  appartient(Object élément) ;
        // retourne la position du premier objet dont l'adresse est égale à
        // celle de élément  ou -1 sinon.
}
```

- **(2 points)** Ecrire la classe **VecteurObjectImpl** qui est une implémentation de l'interface **VecteurObject** et qui fixe la taille du Vecteur au moment de sa construction. Les exceptions n'ayant pas été traitées les cas d'erreurs seront simplement signalés par **System.out.println(« ERREUR »)** ;

```
public class Vecteur
{
    private int taillePhysique;
    private int indiceMaximum;

    private Object[] données;


    public Vecteur(int taille)
```

```
    {
                données = new Object[taille];

                indiceMaximum = -1;
                taillePhysique = taille;
    }

    public void ajouter(int indice, Object element)
    {
                if(indice >= taillePhysique || indice < 0)
                   System.out.println("ERREUR");
                else{
                   données[indice] = element;
                   if((element != null) && (indice > indiceMaximum))
                         indiceMaximum = indice;
                }
    }

    public void supprimer(int indice)
    {
                if(indice >= taillePhysique || indice < 0)
                   System.out.println("ERREUR");
                else{
                   données[indice] = null;
                   if(indiceMaximum == indice)
                         {
                            indiceMaximum = -1;
                            for(int i = taillePhysique; i >= 0; i--)
                                  if(données[i] != null)
                                     {
                                           indiceMaximum = i;
                                           return;
                                     }
                         }
                }
    }

    public int longueur()
    {
                return indiceMaximum;
    }

    public int appartient(Object element)
    {
                for(int i = 0; i < taillePhysique; i++)
                   if(element == données[i])
                         return i;

                return -1;
    }
}
```

(2 points) Dans ce cas, VecteurObject est un conteneur de références d'objet. Quelles seraient les modifications à apporter pour en faire un conteneur d'états d'objets. Expliquez les modifications.

Il faut utiliser le service clone lorsqu'on insère un objet et le service equals lorsqu'on recherche un objet dans le vecteur. Le service clone étant protected dans la classe Object il est nécessaire de déclarer comme Cloneable l'objet que l'on insère dans le vecteur.

```
public class Vecteur
{
   private int taillePhysique;
   private int indiceMaximum;

   private Object[] données;


   public Vecteur(int taille)
   {
     données = new Object[taille];

     indiceMaximum = -1;
     taillePhysique = taille;
   }

   public void ajouter(int indice, Cloneable element)
   {
     if(indice >= taillePhysique || indice < 0)
        System.out.println("ERREUR");
     else{
        données[indice] = element.clone();
        if((element != null) && (indice > indiceMaximum))
                indiceMaximum = indice;
     }
   }

   public void supprimer(int indice)
   {
     if(indice >= taillePhysique || indice < 0)
        System.out.println("ERREUR");
     else{
        données[indice] = null;
        if(indiceMaximum == indice)
                {
                   indiceMaximum = -1;
                   for(int i = taillePhysique; i >= 0; i--)
                        if(données[i] != null)
                           {
                                indiceMaximum = i;
                                return;
                           }
                }
     }
   }
```

```
    public int longueur()
    {
      return indiceMaximum;
    }

    public int appartient(Object element)
    {
      for(int i = 0; i < taillePhysique; i++)
         if(element.equals(données[i]))
                  return i;

      return -1;
    }
}
```

Exercice 2.

(0,5 point) Donner une définition du concept de surcharge en Java. Justifiez son intérêt. La surcharge est la possibilité le même nom dans une même classe pour identifier des services différents. Pour que la surcharge soit licite, il faut que les services de même nom différent par le nombre ou le type des paramètres. Le type de retour n'est pas distinctif dans la surcharge. L'intéret de la surcharge est d'améliorer la compréhension d'une classe, puisque nous pouvons utiliser le même nom pour des services qui ont la même siginification.

- **(0,5 point)** Donner une définition du concept de redéfinition en Java. Justifiez son intérêt.
- La redéfinition est la possibilité d'utiliser la même signature dans un type et sans un sous-type. La fonction redéfinit au niveau du sous type peut changer, spécialiser le comportement d'un service du sur type. La redéfinition couplée avec l'algorithme de sélection permet d'obtenir la propriété suivante : Un objet préserve son comportement indépendamment du type qui le déclare.

- **(0,5 point)** Donner une définition de type déclaré et de type réel. Précisez cette notion sur un exemple. Le type déclaré est le type de la variable qui est utilisé pour référencé un objet. Le type réel d'un objet est le type de la classe qui l'a instancié.
  Ex: Animal a = new Cochon(). Le type déclaré de a est Animal, le type réel de a est Cochon.

- **(1,5 points)** Rappelez les deux étapes de l'algorithme d'exécution des services d'instances en Java.
  - Résolution de la surcharge. En partant du **type déclaré** on collecte l'ensemble des services qui ont le nom recherché sur l'ensemble du haut de l'arbre d'héritage. On supprime celles qui n'ont pas la bonne arité et qui ne sont pas compatibles avec l'appel. Si il n'en reste aucune, il y a une erreur de compilation. Ensuite parmis les candidates restantes ont évalue le coût de conversion. Si il y a plusieurs fonctions de coût minimal, il y a une erreur de compilation. Sinon il n'y a qu'une seule signature qui est conservée, la phase de compilation est correcte.
  - Exécution du service. En partant du type réel de l'objet, on recherche dans le haut de l'arbre d'héritage un service qui correspond exactement à la signature sélectionnée à l'étape 1. On exécute le premier service rencontré qui a exactement cette signature.

- **(3 points)** Soient les hiérarchies de classes suivantes :

```
class A {}
class B extends A{}
class C extends B{}

class X {
        public void m(B b){ System.out.println(« X  param B ») ;
        public void m(A a){ System.out.println(« X  param A ») ;
}
class Y extends X {
        public void m(C c) {System.out.println(« Y  param C ») ;
        public void m(B b) {System.out.println(« Y  param B ») ;
}
class Z extends Y{
        public void m(B b) {System.out.println(« Z  param  B ») ;
        public void m(A a) {System.out.println(« Z  param  A») ;
}
```

Pour le code ci-dessous, donner pour chacun des 9 appels, les résultats donnés par chacune des étapes de l'algorithme d'exécution des services d'instances en Java.

```
public static void main (String [] argv)
{
        C c = new C() ; B b = c ;  A a = c ;
        Z z = new Z() ; Y y = z ;  X x = z ;
        x.m(a) ; // appel 1   y.m(a) ; // appel 2   z.m(a) ; // appel 3
        x.m(b) ; // appel 4   y.m(b) ; // appel 5   z.m(b) ; // appel 6
        x.m(c) ; // appel 7   y.m(c) ; // appel 8   z.m(c) ; // appel 9
}
```

appel 1:
Etape 1: signature sélectionnée m(A a)
Etape 2: Resultat Z param A
appel 2
Etape 1: signature sélectionnée m(A a)
Etape 2: Resultat Z param A
appel 3
Etape 1: signature sélectionnée m(A a)
Etape 2: Resultat Z param A
appel 4
Etape 1: signature sélectionnée m(B b)
Etape 2: Resultat Z param B
appel 5
Etape 1: signature sélectionnée m(B b)
Etape 2: Resultat Z param B
appel 6
Etape 1: signature sélectionnée m(B b)
Etape 2: Resultat Z param B
appel 7
Etape 1: signature sélectionnée m(B b)
Etape 2: Resultat Z param B
appel 8
Etape 1: signature sélectionnée m(C c)
Etape 2: Resultat Y param C
appel 9

Etape 1: signature sélectionnée m(C c)
Etape 2: Resultat Y param C

Exercice 3.

Soit une partie du code de l'interface Point2D

```
public interface Point2D
{
        public Point2D dupliquer() ;
        // crée un nouveau Point2D dans le même que le sélecteur.

         public boolean  equals(Object o) ;
        // Compare si deux instances de Point2D sont dans le même état

         public void deplacer(double x, double y) ;
         // déplace le sélecteur aux coordonnées cartésiennes (x,y)
   }
```

Une interface **FormeCentrée** propose deux services qui sont ***déplacerCentre*** et ***dessiner***. Cette interface **FormeCentrée** est implémentée partiellement par une classe abstraite **FormeCentréeImpl.** Cette classe abstraite contient une variable d'instance

protected Point2D centreForme ;

Elle factorise le code du service ***déplacerCentre*** qui consiste à déplacer la variable centreForme. Dans cette classe abstraite le service ***dessiner*** et abstrait. Les classes **Carré** et **Cercle** sont deux implémentations de **FormeCentrée** qui héritent de **FormeCentréeImpl** et qui implémentent leur propre service d'instance ***dessiner***. On se contentera pour dessiner un Carré (resp un Cercle) d'afficher sur la sortie standard « je suis un carré (resp. un cercle) » .

- **(1 point)** Ecrire les classes de **FormeCentréeImpl, Carré et Cercle. ATTENTION**, Le centre d'une FormeCentrée ne peut être déplacé que par un service de FormeCentrée ou des sous classes de FormeCentrée.
- (**1 point**). Redéfinir le service d'instance **equals** sur la hiérarchie de classe de **FormeCentrée**.

```
public abstract FormeCentreeImpl implement FormeCentree
{
   protected Point2D centreForme;

   protected FormeCentreeImpl(Point2D p)
    {
      centreForme = p.dupliquer();
    }

   public deplacerCentre(double x, double y)
    {
      centreForme.deplacer(x,y);
    }
```

```
    public boolean equals(Object o)
      {
        if( o instanceof FormeCentreImpl)
          {
            FormeCentreeImpl tmp = (FormeCentreeImpl) o;
            return tmp.centreForme.equals(centreForme);
          }
        return false;
      }
}

public class Carre extends FormeCentreeImpl
{
    protected double cote;

    public Carre(Point2D ce, double c)
    {
      super(ce);
      cote = c;
    }

    public void dessiner()
    {
      System.out.println("Je suis un CarrÈ");
    }

    public boolean equals(Object o)
      {
        if( o instanceof Carre)
          {
            Carre tmp = (Carre) o;
            return super.equals(tmp) && tmp.cote == cote;
          }
        return false;
      }
}


public class Cercle extends FormeCentreeImpl
{
    protected double rayon;

    public Cercle(Point2D ce, double r)
    {
      super(ce);
      rayon = r;
    }

    public void dessiner()
    {
      System.out.println("Je suis un Cercle");
```

```
    }

    public boolean equals(Object o)
     {
        if( o instanceof Cdercle)
          {
             Cercle tmp = (Cercle) o;
             return super.equals(tmp) && tmp.rayon  == rayon;
          }
        return false;
     }
}
```

Une interface **Animation** propose trois services d'instances qui sont ***reprendreAnimationAuDébut****,* ***avancerAnimation****,* ***animationTerminée***. L'interface Animation est :

```
public interface Animation
{
     public void reprendreAnimationAuDébut() ;
     public void avancerAnimation() ;
      public boolean animationTerminée() ;
}
```

Cette interface **Animation** est implémentée par deux classes qui sont **AnimationConstanteImpl** et **AnimationAccéléréeImpl.** Les services de ces deux classes font évoluer un point2D qu'elles reçoivent en paramètre au moment de leur instanciation et qu'elles stockent par référence.

En reprenant les spécifications du début de l'exercice, on s'intéresse maintenant à composer les deux hiérarchies.

- **(1 point)** On s'intéresse à la définition du type **CarréAnimé** qui est un **Carré** et une **Animation.** Expliquer pourquoi **CarréAnimé** ne peut pas être une classe **instanciable**. Donner le code de la classe abstraite **CarréAnimé** dans ce cas. Précisez et justifiez pourquoi certains services de cette classe sont nécessairement abstraits.

  Un carréAnimé est nécessairement un Carré, mais on ne peut pas préciser qu'elle animation permet au carréAnimé de fonctionner. Donc la classe CarréAnimé est une classe abstraite qui ne peut pas implémenter les services de l'interface Animation. Il faudrait que les spécifications sur CarréAnimée soit plus précise. Comme l'interface Animation n'est pas Cloneable ou ne possède pas de service dupliquer on ne peut dans ce cas utiliser la délégation.

  ```
  abstract public class CarreAnime extends Carre implements Animation
  {
       public abstract void reprendreAnimationAuDébut();
       public abstract void avancerAnimation();
       public abstract boolean animationTerminÈe();
  }
  ```

- **(5 points)** On voudrait maintenant définir une classe **CarréAnimationConstante.**

1. **(1 point)** Rappelez la définition de l'héritage de type et de ses propriétés. Rappelez la définition de l'héritage de code et de ses propriétés. Existe t-il en Java de l'héritage multiple de code ?
   - L'héritage de type se traduit par la relation « est une sorte de » ou « est un ».Les deux propriétés sont:
     1. On peut référencer par une variable du sur type une variable du sous-type, (principe de substitution)
     2. L'interface fonctionnelle publique du sur-type est incluse dans l'interface fonctionnelle publique du sous type.
   - L'héritage de code, c'est d'abord de l'héritage de type plus la possibilité d'utiliser partiellement ou totalement au niveau du sous type le code du sur type.
   - En java, il n'y a que de l'héritage simple de code. C'est à dire qu'une classe ne peut hériter du code que d'une seule classe.

   (1 point) Expliquez pourquoi en respectant les spécifications précédentes, il est impossible de définir une classe **CarréAnimationConstante** qui soit à la fois une sorte de **Carré** et d'**AnimationConstanteImpl**. Carré et AnimationConstanteImpl sont deux classes qui définissent un type. Comme en Java, il n'y a pas d'héritage multiple de code et comme il n'y a pas une interface qui représente soit le type Carré soit le type AnimationConstante et de Carré.

2. **(3 points)** On ajoute une interface **AnimationConstante** qui est un sur type de **AnimationConstanteImpl** et sous type de **Animation**. Donner dans ce cas, la définition Java du type **CarréAnimationConstante** qui est un sous type de **Carré** et de **AnimationConstante.** Proposer alors une implémentation de telle sorte que la classe **CarréAnimationConstante** soit instanciable. **Dans ce cas, l'animation se chargera de déplacer le centre du Carré.**

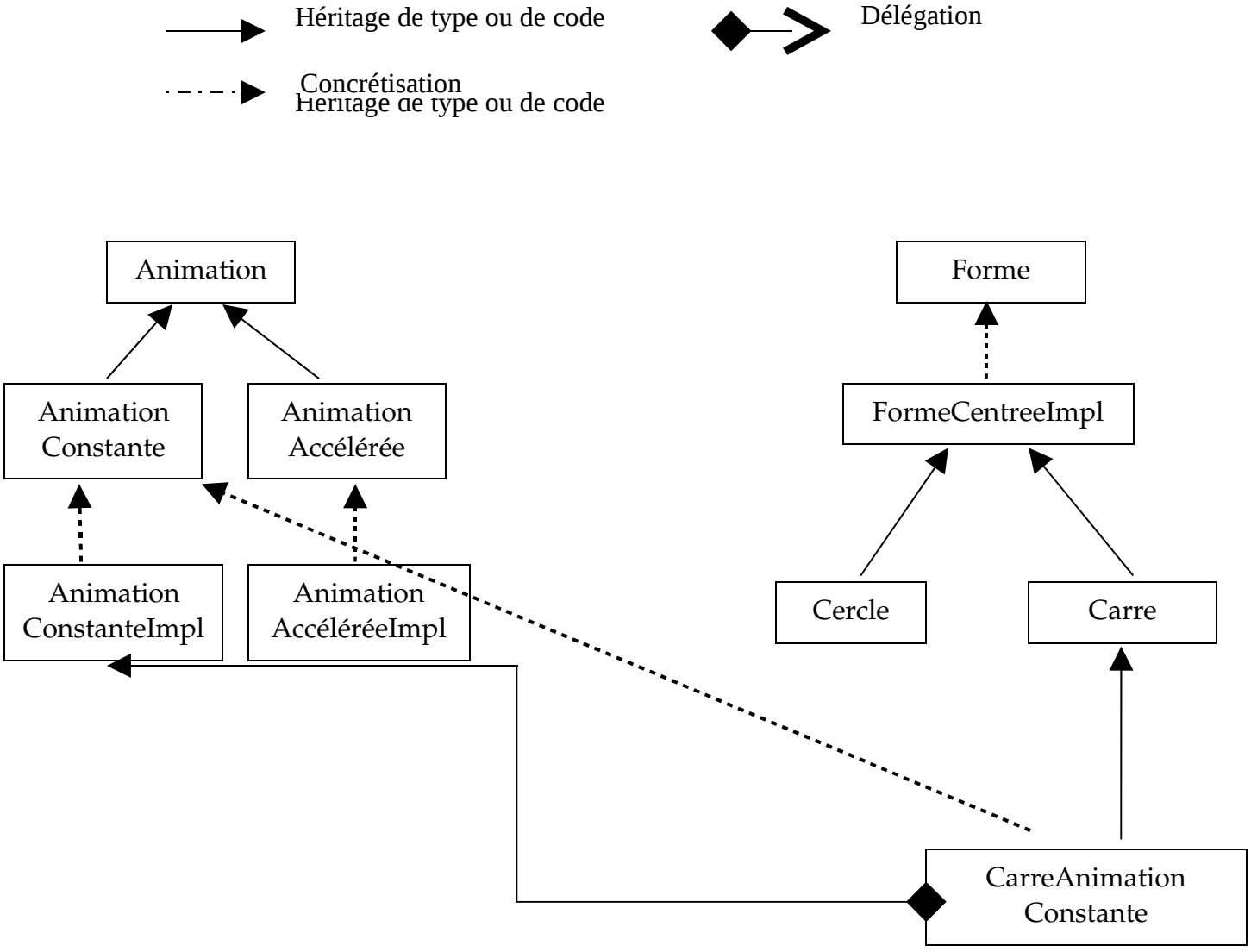
```
public class CarreAnimation extends Carre implements AnimationConstante
{
        AnimationConstanteImpl delegAnimation ;
        Public CarreAnimation(Point2D p, double cote)
        {
                Super(p,cote) ;
                delegAnimation = new AnimationConstante(centreForme) ;

        }

        public  void reprendreAnimationAuDébut()
        {
                delegAnimation. reprendreAnimationAuDébut() ;
        }
        public  void avancerAnimation()
        {
                delegAnimation. avancerAnimation().
        }
         public  boolean animationTerminée()
        {
                return delegAnimation. animationTerminée()

        }
}
```

- **(1 point).** En vous inspirant de la question précédente ajoutez les classes et interfaces qui vous semblent nécessaires pour pouvoir insérer dans les spécifications les classes CarreAnimationAccélérée, CercleAnimationConstante, CercleAnimationAccélérée. Donnez un schéma qui présente les relations (héritage de type, héritage de code, concrétisation, délégation) entre tous les types de cet exercice et ceux que vous avez ajoutés.



- **(1 point)** Est ce qu'il est possible en utilisant la délégation d'écrire une classe **CarreAnimationImpl** qui fournirait de la factorisation pour code les classes **CarreAnimationConstante** et **CarreAnimationAccelérée.** Donnez l'implémentation de cette classe.

  Oui il faut que cette classe prenne dans un constructueur une Animation.

```
abstract public class CarreAnimationImpl extends Carre implements AnimationConstante
{
    AnimationConstanteImpl delegAnimation ;
   protected CarreAnimationImpl(Point2D p, double cote, AnimationConstant anim)
    {
        Super(p,cote) ;
        delegAnimation = anim ;

    }

    public  void reprendreAnimationAuDébut()
    {
        delegAnimation. reprendreAnimationAuDébut() ;
    }
    public  void avancerAnimation()
    {
        delegAnimation. avancerAnimation().
    }
     public  boolean animationTerminée()
    {
        return delegAnimation. animationTerminée()

    }
}
```

FIN