DISVE Licence	**ANNEE UNIVERSITAIRE 2007/2008** **1ère Session de Printemps** **Parcours** : CSB4, MHT4 **UE** : INF252 **Epreuve** : Programmation2 **Date** : 6 Mai 2008 **Heure** : 8h **Durée :** 1h30 **Documents :** Tous Documents autorisés Le document fait 4 pages. **Epreuve de :** JP.Domenger	

Problème.

La plus part des questions de ce problème sont indépendantes. Elles peuvent être traitées sans avoir répondu aux autres questions simplement en utilisant les informations présentes dans le texte. De plus, ceraines questions sont une application directe du cours et des TD. Prenez la peine de lire le texte dans son intégralité.

La clarté, la concision et la précision des réponses seront fortement appréciées par le correcteur. Le barème proposé n'est donné qu'à titre indicatif.

Pour ce problème on considérera que la classe Point2DCartésien existe, nous rappelons ci-après son interface fonctionnelle publique, le code n'est pas utile pour ce problème.

```
public class Point2DCartésien implements Cloneable {
        private double abscisse ;
        private double ordonnée ;

        static public double distance(Point2DCartesien p, Point2DCartesien q) {…//code//}
        public Point2DCartesien( double abs, double ord) {…// code ……}

        double donneAbscisse(){…. // code….}
        double donneOrdonnée(){….// code….}

        public void translation(double dx, double dy) {……. // code…….}

        public boolean equals(Object o) {……. // code ……} ;
        public String toString() {……. // code ……};
}
```

Question 1 (1 point) : Rappeler les propriétés du service **Object clone()** dans la classe **Object**. Ecrire et justifier le code de ce service pour la classe **Point2DCartésien**.

Le service clone a pour objectif de créer un objet identique à celui qui a en charge la réalisation du service. Le comportement par défaut qui est donné au niveau de la classe object est :

1. **allocation mémoire nécessaire**
2. **Copie bit à bit.**

3. **Préservation du type réel de l'objet.**

Les variables d'instances de la classe Point2DCartésien sont des types primitifs, dans ce cas la copie bit à bit est suffisante. Le code de clone est donc :

```
public Object clone()
{
        return super.clone() ;
}
```

On définit l'interface **Segment**:

```
public interface Segment  implements Cloneable
{
        public Point2DCartésien donneOrigine() ;
        public Point2DCartésien donneExtrémité() ;

        public double donneLongueur() ;
        public  Point2DCartesien donneMilieu() ;
        public void     translation(double dx, double dy) ;

        public boolean equals(Segment s);
        public String toString() ;
        public Object clone() ;

}
```

Question 2 (**3 points)** : On considère que la classe **SegmentImpl** est une implémentation de l'interface **Segment**, telle que la structure d'une instance de la classe **SegmentImpl** est constituée de deux **Point2DCartésien** qui sont l'**origine** et l'**extrémité** du segment. Ecrire le code de la classe **SegmentImpl** en utilisant **Point2DCartésien**. Ajouter les constructeurs qui vous semblent judicieux pour cette classe. **ATTENTION**, ***on ne veut pas que l'origine et l'extrémité d'un objet de type réel* SegmentImpl *soient modifiées à l'extérieur de la classe* SegmentImpl.**

```
public class SegmentImpl implements Segment
{
   private Point2DCartesien origine;
   private Point2DCartesien extremite;

   public Point2DCartésien donneOrigine(){
        return origine.clone();
   }
   public Point2DCartésien donneExtrémité(){
```

```
        return extremite.clone():
    }
    public double donneLongueur(){
        return Point2DCartesien.distance(origine, extremité);
    }
    public  Point2DCartesien donneMilieu()
    {
        double dx = extremite.donneAbscisse() - origine.donneAbscisse();
        double dy = extremite.donneordonnee() - origine.donneOrdonneé();

        return new Point2DCartesien(dx/2, dy/2);
    }
    public void translation(double dx, double dy)
    {
        origine.transalation(dx,dy);
        extremite.translation(dx,dy);
    }

    public boolean equals(Segment s)
    {
        return (origine.equals(s.donneOrigine()) &&
                extremite.equals(s.donneExtremite())) ||
           (origine.equals(s.donneExtremite()) &&
            extremite.equals(s.donneOrigine()));
    }

    public String toString(){
        String string = "Je suis un segment" +
           "Mon origine est" + origine.toString();
           "Mon extremite est" + extremité.toString();

           return string;
    }

    public Object clone(){
        SegmentImpl s = (SegmentImpl) super.clone();
        s.extremite = extremite.clone();
        s.origine   = origine.clone();

        return s;
    }
  public SegmentImpl(Point2DCartesien o, Point2DCartesien e){
        origine = o.clone();
        extremite = e.clone();
    }
}
```

Question 3 (3 points) : Qualifier en terme de redéfinition et/ou de surcharge le service **boolean equals(Segment s)**. Donner des explications.
Soit le code suivant :

```
Point2DCartésien o = new Point2DCartésien(0,0) ;
Point2DCartésien e = new Point2DCartésien(1,1) ;

Object s1 = new SegmentImpl(o,e) ;
Object s2 = new SegmentImpl(o,e) ;

if( s1.equals(s2))
        System.out.println("Les segments sont égaux");
else
        System.out.println("Les segments sont différents");
```

Quel est le type réel de s1 ? Quel est le type déclaré de s1? Expliquer quel est le résultat produit par le code ci-dessus en détaillant les 2 étapes de l'algorithme de sélection des services sur cet exemple. Est ce que le résultat vous convient et pourquoi? Quelles sont les modifications à apporter à l'interface **Segment** et à la classe **SegmentImpl**. Ecrire le code nécessaire.

Il s'agit d'une surcharge du service **boolean equals(Object). En effet, toute classe hérite de Object, comme le service boolean equals(Object) est publique dans la classe Object il est implicitement présent dans la classe SegmentImpl. On a donc bien le même nom de service avec des signatures de fonctions différentes dans le contexte de la classe SegmentImpl.**

Le type réel de s1 est segmentImpl, le type déclaré est Object.

L'algorithme de résolution procède en deux étapes :

- **En fonction du type déclaré, exécution de l'algorithme de résolution de la surcharge qui retient une signature unique compatible avec l'appel.**
- **En fonction du type réel exécution du premier service correspondant à la signature sélectionnée à l'étape précédente.**

L'étape de résolution de la surcharge se fait dans le contexte du type déclaré. Une seule méthode existe c'est celle de Object. La signature sélectionnée de la fonction est donc

boolean equals(Object).

Après vérification de l'arité et de la compatibilité, c'est cette méthode qui est sélectionnée.

Ensuite en partant du type réel, à savoir SegmentImpl, on regarde si il existe un service qui a pour signature

boolean equals(Object).

La réponse est non, la seule qui existe est

boolean equals(Segment)

On remonte donc, et on finit par exécuter celle de la classe Object. Le résultat ne convient pas car, on compare les adresses des deux segments qui sont différentes et le résultat produit est donc :

"Les segments sont différents"

Pour obtenir un résultat correct, il faut redéfinir le service boolean equals(Object) dans la classe SegmentImpl dont le code est :

```
public boolean equals(Object o) {

        if( ! (o instanceof(Segment) ) return false;
        else
                return equals((Segment) o);
}
```

Maintenant on considère une nouvelle implémentation de l'interface **Segment** qui est donnée par la classe **SegmentImplNouvelle** qui est **instanciable.**

Question 4 (1 point) : Rappeler la/les relations qui existent entre une interface et une classe qui l'implémente. A quelle condition une classe qui implémente une interface est-elle instanciable ? Qu'est ce qu'une classe abstraite et à quoi peut-elle servir ?

La relation entre une classe et une interface est une relation de concrétisation, ou d'implémentation. Elle implique la relation d'héritage de type entre la classe et son sur-type l'interface. La classe a en charge de fournir une implémentation partielle ou totale des services définit dans le l'interface fonctionnelle publique. Exemple de déclaration :

Interface I{……..}

class A implements I{…….. // code pour tous les services déclarés dans I}

Pour qu'une classe qui implémente une interface soit instanciable, elle doit fournit une implémentation pour l'ensemble des services déclarés dans l'interface fonctionnelle publique de l'interface. Sinon c'est une classe abstraite.

Une classe abstraite définit une interface fonctionnelle publique, mais elle ne fournit pas l'intégralité du code pour l'implémentation de ses services. L'intérêt des classes abstraites est de fournir une factorisation de code pour ses sous classes, sans pour autant être instanciable directement.

```
abstract class B implement I { ….
        abstract public unServiceNondefinit();
        …..
}
```

Question 5 (2 points) : Est ce que l'ajout de cette nouvelle implémentation de l'interface **Segment,** nécessite de modifier l'implémentation de la classe **SegmentImpl** ? ***Doit-on par exemple, modifier le code du service equals ?*** Est ce qu'il existe du code qui peut-être factorisé entre **SegmentImpl** et **SegmentImplNouvelle ? Si la réponse est oui, proposez une solution qui permet d'éviter cette duplication de code. L'implémentation totale ne la classe n'est pas demandée.**

Tout dépend de la manière dont on a écrit le service equals à la deuxième question. Mais maintenant il faut pouvoir comparer un objet de type reel SegmentImpl avec un objet de type réel SegmentImplNouvelle. C'est le cas pour notre implémentation, le nouveau code est donc :

```
public boolean equals(Segment s)
   {
        return (donneOrigine.equals(s.donneOrigine()) &&
                donneExtremite().equals(s.donneExtremite())) ||
           (donneOrigine().equals(s.donneExtremite()) &&
            donneExtremite().equals(s.donneOrigine()));
   }
```

Dans ce cas, on utilise que des services de l'interface fonctionnelle de l'interface Segment. Ce code ne dépend d'aucune implémentation il peut donc être factorisé.

```
abstract class SegmentAbstracFacto implements Segment
{

        public boolean equals(Segment s) {…… meme code que au dessus..}
        public boolean equals(Object o) {…… comme avant…}
        public String toString() {……. Même code qu'a la question 2..}

        // LES AUTRES SERVICES SONT ABSTRACT….
}
class SegmentImpl extends SegmentAbstractFacto {…….}

class SegmentImplNouvelle extends SegmentAbstractFacto {…….}
```

On veut maintenant écrire une **classe LigneBrisée.** Une ligne brisée est une suite de segments telle que pour deux segments successifs s1 et s2, l'extrémité de s1 est égale à l'origine de s2. Comme pour la question 2, on veut que les instances de ligne brisée possèdent leurs propres instances de **Segment** qui ne peuvent être référencées par l'extérieur.

La première version de **LigneBrisée** est basée sur une implémentation avec un tableau de taille fixe de segments. Une implémentation partielle et naïve pourrait être la suivante :

```
public class LigneBrisée {
        Segment [] ligne = new Segment[10] ;
        int indiceCourant = -1 ;
        int tailleMax = 10 ;

        public LigneBrisée(Segment s)
        {
                ligne[++indiceCourant] = s.clone() ;
        }

        public ajoutSegment(Segment s1)
        {
```

```
                if( ! ligne[indiceCourant].donneExtremite.equals(s1.donneOrigine())){
                        System.out.println(« Le segment n'est pas valide ») ;
                        System.exit() ;

                }

                if(indiceCourant + 1 = tailleMax) {
                        System.out.println(« Un débordement a eu lieu ») ;
                        System.exit() ;
                }
                ligne[++indiceCourant] = s1.clone() ;

        }
}
```

Question 6 (2 points). On veut modifier cette implémentation de la classe **LigneBrisée** pour gérer les erreurs en utilisant les exceptions. Quels sont les deux types d'exception qui peuvent exister en Java ? L'exception liée au dépassement de tableau est représentée par la classe **DepassementTableauException** et celle liée à la condition de non contiguïté par la classe **NonContigueException.** Donner la déclaration de ces deux classes d'exception **(pas le code)**, précisez quel est le sur-type de chaque classe. Modifier l'implémentation de la classe **LigneBrisée** ainsi que la signature de **ajoutSegment** (si il y a lieu) pour tenir compte des conditions.

Les exceptions d'utilisation : Ces exceptions doivent obligatoirement être déclarées dans la signature de la fonction. En effet, elles documentent l'utilisation de la fonction. Elles peuvent survenir lorsque le service ne peut être effectué correctement. Par exemple, les valeurs de paramètres ne sont pas correctes. Ou bien l'objet qui doit effectuer le service n'est pas dans un état correct. Ces exceptions font partie du service, et même si l'implémentation est modifiée, elles ne devraient pas être remise en cause. On parle aussi parfois d'exception d'interface. C'est le cas pour NonContigueException.

Les exceptions d'implémentation. Ces exceptions sont liées à une implémentation particulière d'un service. Si l'implémentation change, alors l'exception peut-être caduque. Ces exceptions n'ont pas à être déclarées dans la signature de la fonction. Certaines sont générées automatiquement par java pendant l'exécution du programme, il s'agit par exemple des débordements de tableaux, des divisions par zéro. Toutes les exceptions qui sont liées à des ressources physiques, mémoire, descripteur de fichiers, sont des exceptions d'implémentation

Les exceptions d'implémentation héritent du type de Error ou de RuntimeException. Toutes les exceptions qui ne sont pas un sous type de Error ou de RuntimeException sont des exceptions d'utilisation. Cest le cas pour DepassementTableauException.

Les déclarations sont :

class NonContigueException extends Exception {…….}
class DepassementTableauException extends Error {……}

```
public ajoutSegment(Segment s1) throws NonContigueException
        {
                if( ! ligne[indiceCourant].donneExtremite().equals(s1.donneOrigine())){
                   throw new NonContiguException() ;
                }

                if(indiceCourant + 1 = tailleMax) {
                   throw new DepassementTableauException() ;
                }
                ligne[++indiceCourant] = s1.clone() ;

        }
```

On ajoute maintenant une nouvelle spécialisation de la ligne brisée. Cette classe est appelée **LigneBriséeContrainte**. Un objet de type **LigneBriséeContrainte** est compatible avec le type **LigneBrisé**, mais il y a condition supplémentaire
« *une ligne brisée contrainte ne s'auto-intersecte pas en dehors des points de raccordement* ».

C'est à dire que deux segments ne peuvent s'intersecter que lorsque ils se succèdent et seulement sur leur unique point commun (l'extrémité du premier segment et l'origine du suivant).

Cette extension nécessite de disposer de deux nouveaux services :

- **estParallèle** retourne un booléen qui indique si deux segments sont parallèles,
- **intersection** retourne une référence à un objet de type Point2DCartésien, qui est **null** si il n'y pas d'intersection et qui sinon retourne le point d'intersection de deux segments.

Question 7 (2 points). On ne demande pas d'écrire le code de ces deux services. Pour chacun des deux services précédents **indiquer et justifier** si ils doivent être déclarés comme un service de la classe de **Segment**, un service de classe d'une autre classe, ou service d'instance de **Segment**. Pour chacune de vos propositions donnez la déclaration.

Les deux services sont symétriques, il n'y a aucune raison de les distinguer par rapport à un segment ou à un autre. On ne demande pas à un segment de dire si il intersecte ou si il est parallèle à un autre segment. On prend deux segments et on vérifie une propriété sur ces deux segments. Il faut donc que ces deux services soient des services de classe et non des services d'instances. On peut les mettre à l'extérieur de l'interface segment car on n'a pas besoin d'accéder à des informations d'implémentation particulière. Tout le code peut s'écrire en utilisant les services donneExtrémité et donneOrigine.

Question 8 (1 point). Si la condition de non intersection n'est pas vérifiée au moment de l'ajout une exception du type **AutoIntersectionException** sera levée. Donner la déclaration de cette classe d'exception. Faut il modifier la déclaration de la classe **NonContigueException** ? Donner les signatures (y compris la déclaration d'exception) des services **ajoutSegment** des classes **LignesBrisée** et **LigneBriséeContrainte.**

Il faut que AutoIntersectionException et NonContiguException soit unifiable sur un même sur-type. Car la classe LigneBriséeContrainte et un sous type de LigneBrisée.

On considère alors un type LigneBriséeException qui est un sous type de Exception.
class LigneBriséeException extends Exception {…….}
class NonContigueException extends LigneBriséeException {…….}
class AutoIntersectionException extends LigneBriséeExcpetion {…….}

Dans la classe LigneBrisée
public ajoutSegment(Segment s1) throws NonContigueException, LigneBriséeException

Dans la classe LigneBriséeContrainte
public ajoutSegment(Segment s1) throws NonContigueException, AutoIntersectionException, LigneBriséeException

Question 9 (2 points). Rappeler le tableau de visibilité des variables en Java. Si on ne change pas la déclaration des variables de la classe **LigneBrisée** où doit être implémenter la classe **LigneBriséeContrainte** si on veut accèder à l'implémentation de **LigneBrisée ?** Sinon quelle peut-être l'autre solution pour accéder à son implémentation depuis la classe **LigneBriséeContrainte** ?

	Même classe	Sous classe	Même package	Extérieur
public	OUI	OUI	OUI	OUI
protected	OUI	OUI	OUI	NON
rien	OUI	NON	OUI	NON
private	OUI	NON	NON	NON

Il faut soit que LigneBrisée et LigneBriséeContrainte soient dans le même package, car les variables d'instances sont préfixées par le mot rien. Leur accès est donc limité au package.
Si on ne veut pas ces deux classes soient dans le même package, il faut que les variables d'instances de LigneBrisée soient suffixées protected.

Question 10 (3 points). En considérant l'implémentation naïve pour la classe **LigneBrisée**. Ecrire l'intégralité du code de la classe **LigneBriséeContrainte**. En tenant compte des exceptions et en minimisant la duplication de code.

```
public class LigneBriséeContrainte extends LigneBrisée {

public LigneBriséeContrainte(Segment s) {
        super(s) ;
}
```

```
public ajoutSegment(Segment s1) throws NonContigueException, AutoIntersectionException,
LigneBriséeException
        {

                for(int i = 0 ; i < indiceCourant ; i++) {
                        if(!GEOLIB.estParralle(s1, ligne[i])){
                            Point2DCartesien tmp = GEOLIB.intersection(s1,ligne[i]) ;
                            if(tmp != null)
                                throw new AutoIntesectionException() ;
                         }
                        if(s1.equals(ligne[i])
                            throw new AutoIntesectionException() ;
                }

                return super.ajoutSegment(s1);
          }
```

Question 11 (2 points). Modifier l'implémentation naïve de la classe LigneBrisée de telle façon que les limitations dues à la taille mémoire du tableau n'existe plus. Vous pouvez considérer l'existence de toutes classes qui vous semblent intéressantes.

Concernant la question suplémentaire, on peut utiliser une liste simplement chainée d'object. Le code est alors le suivant.

```
public class LigneBrisée {

          ListeObject l = new ListeObject() ;

          public LigneBrisée(Segment s)
          {
                l.addLast(s.clone()) ;
          }
        public ajoutSegment(Segment s1) NonContigueException, LigneBriséeException
          {
                Segment tmp =(Segment) l.getLastElement() ;

                if( ! tmp.donneExtremite().equals(s1.donneOrigine()))
                        throw new NonContiguException() ;

                l.addLast(s1.clone()) ;
          }
}

public class LigneBriséeContrainte extends LigneBrisée {

public LigneBriséeContrainte(Segment s) {
          super(s) ;
}
```

```
public ajoutSegment(Segment s1) throws NonContigueException, AutoIntersectionException,
LigneBriséeException
{
		l.goFirst();

		do{

			Segment tmp = (Segment) l.current();
			if(!GEOLIB.estParralle(s1, tmp)){
			     Point2DCartesien tmp = GEOLIB.intersection(s1,tmp) ;
			     if(tmp != null)
			         throw new AutoIntesectionException() ;
			 }
			if(s1.equals(tmp)
			     throw new AutoIntesectionException() ;
			l.next();
		}while(l.hasMore());

		return super.ajoutSegment(s1);
	}
```