Héritage de Code et classe abstraite.

Relation entre Interface-Interfaces, Classe-Interfaces.

Nous avons vu dans le cours précédent la notion de classe et la notion d'interface ainsi que la relation d'héritage de type.

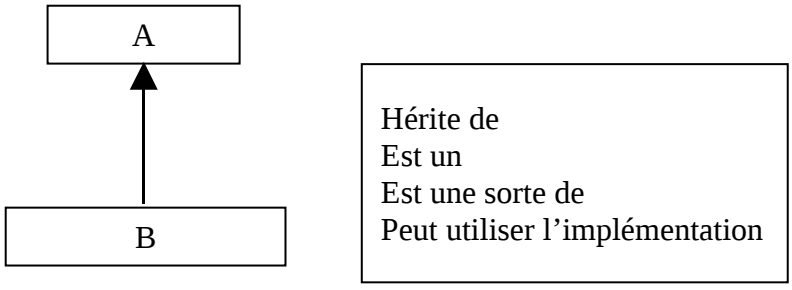
Relation entre une interface et des interfaces: Une interface peut-être une extension d'une ou plusieurs autres interfaces. Dans ce cas, une interface peut-être un sous-type de plusieurs autres types. Dans ce cas, il y a un héritage de type entre le sous-type et le sur-type.

Relation entre une classe et des interfaces: La relation qui existe entre une classe et un ensemble d'interfaces est la relation de concrétisation. Une classe peut réaliser plusieurs interfaces en fournissant l'implémentation des différents services publics déclarés par les interfaces. Dans ce cas, il y a encore un héritage de type entre le type déclaré par la classes et ses sur types qui sont ceux déclarés par les interfaces qu'elle concrétise.

Nous allons maintenant aborder une relation supplémentaire qui est:

Relation entre une classe et une autre classe: La relation qui existe entre deux classes est une relation d'héritage simple.
Elle se déclare en utilisant le mot clef extends.

class A extends B {.....}



Cette relation implique :

1. **Heritage de type entre le type A et le sur-type B.** Cette propriété implique la validité de l'héritage de type. En aucun cas, on ne peut utiliser l'héritage entre classe pour des raisons de facilité d'implémentation. Il faut que dans le domaine du problème, les objets instances de A valident la relation sémantique « Les objets instance de A sont des sortes de B »
2. **Héritage de code, c'est à dire la possibilité pour la classe A de pouvoir utiliser tout au partie du code de la classe B pour implémenter ses propres services.**

Héritage de code.

Considérons par exemple, le code suivant inspiré des TD:

```
public class FormeFacto {
	private Point2D centre;

	public FormeFacto(Point2D p)
	{
		centre = p.creerCopie();
	}

	public void translate(Point2D p, Point2D q)
	{
		centre.translate(p,q);
	}

	public Point2D donneCentre()
	{
		return centre.creeCopie();
	}
}
```

Pour l'instant, la classe FormeFacto est une classe instanciable car elle donne du code à la totalité des services qui sont présents dans son interface fonctionnelle publique.

Maintenant, nous allons nous intéresser à la classe Cercle. La première question à se poser est la suivante: Est ce qu'un cercle est une sorte de FormeFacto particulière qui possède un centre ? Si la réponse à cette question est oui. Alors on peut considérer le type Cercle comme un sous-type de FormeFacto, ce qui valide l'héritage de type et qui permet donc d'utiliser l'héritage entre classe.

```
public class Cercle extends FormeFacto
{
	private double rayon;

	public Cercle(Point2D p, double rayon)
	{
	..... // le code à ajouter.
	}
}
```

Comme un Cercle est une sorte de FormeFacto, il doit posséder tous les attributs d'une FormeFacto c'est à dire un centre. Comme un Cercle est un sous-type de FormeFacto, il doit au moins posséder les services publics de FormeFacto. La première propriété concerne l'héritage de type alors que la deuxième propriété concerne l'héritage de code. Nous allons maintenant aborder les propriétés de l'héritage de code.

Propriété de l'héritage de Code.

1. **Les propriétés de l'héritage de type (principe de substitution et inclusion de IFP) sont impliquées par l'héritage de code.**
2. **A l'intérieur de la structure d'un sous-type, il y a la structure du sur-type.** Cette structure n'est pas toujours accessible, mais elle est toujours présente. Pour construire un

sous type il est donc nécessaire de construire un sur-type, lorsque le sur-type déclare du code. C'est à dire que le sur-type est une classe et non une interface. Cela veut dire que le constructeur du sous-type appelle le constructeur du sur-type. Par défaut, le constructeur du sur type est celui sans paramètres si ce dernier est accessible. Par exemple, si on définit de la manière suivante le constructeur de Cercle.

```
public Cercle(Point2D p, double rayon)
{
// appel implicite au constructeur FormeFacto()
this.rayon = rayon;
}
```

Ce code va générer une erreur car il essaye de construire la partie FormeFacto en utilisant le constructeur sans paramètres hors ce dernier n'est pas accessible. En effet, le fait d'avoir définie le constructeur FormeFacto(Point2D p) masque le constructeur sans paramètre qui est définit par défaut en Java.
Pour pouvoir appeler le constructeur du sur-type depuis le constructeur du sous-type, il faut utiliser la syntaxe **super(.....paramètre)**. Il faut que cet appel soit la première ligne du constructeur du sur-type. Comme il n'y a que de l'héritage simple de code en Java, il ne peut y avoir d'ambiguité avec le mot clef super. Le constructeur de Cercle est alors le suivant:

```
public Cercle(Point2D p, double rayon)
{
super(p) ; // appel explicite au constructeur FormeFacto(Point2D)
this.rayon = rayon;
}
```

On peut même rajouter un autre constructeur en utilisant l'appel à un autre constructeur grâce au mot clef **this.**

```
public Cercle()
        {
                this(new Point2DCartesien(0,0), 1); // appel a Cercle(Point2D,double)
        }
```

Donc maintenant lorsqu'on appelle new Cercle(p,3) on déclenchera un appel à FormeFacto(p).

3. La troisième propriété concerne le partage de code entre le sur-type et le sous type. Le code du sur-type peut-être totalement ou partiellement utilisé par le sous type. Il faut bien que ce code soit accessible (déclaré protected), nous reviendrons par la suite sur la déclaration permettant l'accessibilité. Nous avons vu précédemment que le sous-type contient la structure du sur-type, donc le code qui est définit sur le sur-type peut être utilisé avec le sur-type.
    - Utilisation d'un service déclaré dans le sur-type. Si on reprend la déclaration de Cercle.

```
public class Cercle extends FormeFacto
{
        private double rayon;

public Cercle(Point2D p, double rayon)
```

```
    {
    ..... // le code à ajouter.
    }
}
```

Le service translation n'est pas défini, mais la classe cercle est instanciable donc le code pour faire fonctionner le service translation existe et il doit être accessible lorsqu'on le selectionne en utilisant un cercle. Prenons le code suivant:

```
Cercle c = new Cercle(p,2);
FormeFacto f = c; // principe de substitution
                  // c et f identifie exactement le même objet.
f.translate(.....); // instruction correcte car f est une classe instanciable et donc
                    // tous les services publics sont implémentés.

c.translate(.....); // doit lui aussi fonctionner car c a été instancié.
                    // le codee qui sera utilisé sera celui de FormeFacto.
```

Dans ce cas, l'héritage de code permet d'utiliser l'intégralité d'un service déclaré dans le surtype sans avoir besoin de le déclarer ou de le définir dans le sous-type.

- Utilisation et appel d'un service du sur-type à l'intérieur du code du sur-type.

Rajoutons par exemple le service toString() à la classe FormeFacto

```
public class FormeFacto {
    private Point2D centre;
    public FormeFacto(Point2D p)
    {......}
    public void translate(Point2D p, Point2D q)
    {.......}
    public Point2D donneCentre()
    {......}
    public String toString() {
    return « Je suis une formeFacto » + centre.toString();
    }
}
```

On veut maintenant définir un service toString() au niveau de la classe Cercle, mais on veut que les informations de FormeFacto apparaissent. Il faut donc à l'intérieur du service toString() de Cercle appelée le service toString() de FormeFacto(). Pour appeler le service de nom XX du sur type il faut utiliser la syntaxe suivante :

```
super.XX(....)
```

Nous pouvons maintenant écrire le code du service toString() de la classe Cercle.

```
public class Cercle extends FormeFacto
    {
        private double rayon;

        public Cercle(Point2D p, double rayon)
        {..... }
        public String toString()
        {
           String tmp = super.toString(); // appel a toString() de FormeFacto
```

```
                return « Je suis un cercle de rayon » + rayon + tmp;
            }
        }
```

Le mot clef protected.

Nous avons vu dans les premiers cours que pour contrôler l'acces à l'information (service ou variables). On pouvait utiliser les mots clefs private et public. Les informations private ne sont accessibles qu'à l'intérieur de la classe qui les définit, les informations public sont accessible par tout le monde. La relation d'héritage de code crée un lien particulier entre une sous-classe et sur classe. Il peut-être nécessaire d'accéder depuis la sous classe à une partie de l'implémentation de la surclasse. Sans pour autant que l'information soit visible pour tout le monde. En effet comme une partie du code peut-être partagée par la sous classe, il est parfois nécessaire qu'elle puisse accéder à la structure du sur-type qui la constitue en partie. Pour permettre l'accès à l'information de la sur-classe depuis la sous-classe, il faut utiliser le mot clef protected lorsqu'on déclare l'information.

Par exemple, si la classe Cercle devait accèder à la variable centre de la classe FormeFacto pour la modifier. La variable centre est bien à l'intérieur de la structure d'un objet centre mais elle n'est pas accessible car son accès est privée. Par contre, elle est accessible en lecture en utilisant le service donneCentre(), mais on peut pas la modifier car c'est seulement une copie qui est retournée.

Comme il est bien sûr hors de question de la rendre publique, on doit la déclarer protected. Et elle devient alors publique pour les sous-classes et private pour toutes les autres classes. La déclaration devient alors.

```
public class FormeFacto {
        protected Point2D centre;
.......

}
```

Classe Abstraite.

Nous avons vu qu'une classe est non instanciable si elle ne fournit pas l'intégralité du code pour implémenter l'ensemble de ces services publics. Qu'une interface n'est pas instanciable car elle n'implémente aucun service public. Une classe peut tout de même présenter une implémentation de certains services publics qui peut être factorisée pour l'ensemble de ses sous-classes, mais elle n'est pas capable de fournir l'intégralité de l'implémentation des services. Dans l'exemple sur les animaux, nous avons vu que nous pouvions donner par exemple du code pour le service mange des herbivores, mais nous ne pouvons pas en donner pour le service reproduire, donc cette restriction nous a obligé à transformer Herbivore en interface, et à engendrer de la duplication de code pour le service mange pour tous les animaux concrets qui sont herbivores (vache, poule,...)

Il existe une solution intermédiaire entre la classe instanciable qui donne l'intégralité du code et l'interface qui ne donne aucun cas. Cette solution consiste à définir une classe abstraite. Une classe abstraite est considérée comme une classe du point de vue de l'héritage de code. Cela veut dire qu'une classe ne peut hériter d'une code que d'une classe qu'elle soit abstraite ou non.

Une classe abstraite est non instanciable directement. C'est à dire que l'on ne peut écrire new.....

Mais par contre ces sous-types pourront créer une instance de cette classe. Une classe abstraite fournit l'impl émentation de certains services de l'interface fonctionnelle publique mais pas de tous. Les services implémentés par une classe abstraite servent pour la factorisation de code. Syntaxiquement, une classe abstraite XX avec un service abstrait serviceAbstrait se déclare de la manière suivante:

```
public abstract class XX
{
        protected int tmp1;

        protected XX(int v) // le constructeur est protected car seule les sous classes de XX peuvent
        {                           // l'instancier.
           tmp1 = v;
        }

        public int inc() // un service factorisé
        {
                return ++tmp1;
        }

        abstract public void serviceAbstrait(int x); // déclaration du service abstrait.
}
```

Sur cet exemple, on ne peut pas écrire directement new XX(2) même si le constructeur avait été déclaré public. La classe n'est pas instanciable directement.

Si on veut utiliser cette classe abstraite on peut définir maintenant une de ces sous classes.

```
public class YY extends XX
{
        public YY(int v)
        {
          super(v); // on instancie implicitement un XX en créant un YY
                    // on appelle le constructeur
        }

        public void serviceAbstrait(int x) // implementation du service abstrait de XX
        {
           tmp1 += 2; // on peut accèder à la variable tmp1 qui est protected
        }
}
```

La classe YY donne une implémentation aux services de son interface fonctionnelle publique. Cette interface fonctionnelle comprend le service inc définit par XX et le service serviceAbstrait qui est déclaré dans XX et qui est définit dans YY. La classe YY est donc instanciable. On peut écrire le code suivant :

```
        YY y = new YY(2); // on crée tout de même un XX;
        XX x = y; // principe de substitution

        x.inc();
        x.serviceAbstrait(); on utilise celui définit par YY.
```

Un autre exemple, peut-être de modifier la classe FormeFacto en lui rajoutant le service surface. Ce service est nécessairement abstrait car nous ne connaissons par réellement le type de Forme que nous allons manipuler. Mais par contre, tous les utilisateurs de FormeFacto peuvent appeler le service surface. Le code devient alors :

```
abstract public class FormeFacto {// FormeFacto est maintenant abstraite donc non instanciable.
        protected Point2D centre;

        protected FormeFacto(Point2D p)
        {
                centre = p.creerCopie();
        }

        public void translate(Point2D p, Point2D q) //code factorisé
        {
                centre.translate(p,q);
        }

        public Point2D donneCentre() // code factorisé
        {
                return centre.creeCopie();
        }
        abstract double surface(); // le service surface est abstrait.
}
```

La classe Cercle maintenant pour continuer à être instanciable doit implémenter ce service.

```
public class Cercle extends FormeFacto
{
        private double rayon;

        public Cercle(Point2D p, double rayon  {..... }
        public String toString(){
                                        String tmp = super.toString(); // appel à  toString() de FormeFacto
                                         return « Je suis un cercle de rayon » + rayon + tmp;
                                    }
        public double surface() { // implementation du service abstrait.
                  return Math.PI * rayon*rayon;
        }
}
```

Si nous revenons à l'exemple des Animaux vus précédemment, nous avons constaté qu'il y avait de la duplication de code au niveau des classes concrètes Vache, Poule,..... Par exemple, les services allaite et reproduire sont les mêmes pour tous les mammifères, les services vole et reproduire sont les mêmes pour tout les oiseaux, le service mange et le même pour tous les herbivores....etc. En utilisant les classes abstraites nous pouvons réduire la duplication de code soit pour tous les mammifères et pour tous les oiseaux, soit pour tous les herbivores, les carnivores et les omnivores. Mais on ne pourra pas supprimer toute la duplication de code car il n'y a que de l'héritage simple de code en Java. Il faudra utilisée la délégation.

Nous allons simplement illustré, l'utilisation des classes abstraites et de l'héritage pour factoriser le

code des mammifères et nous allons utiliser la délégation pour factoriser le code des herbivores.

```
Interface Mammifère extends Animal
{.............}

abstract public class MammifèreImpl implements Mammifère // class non instanciable
{
        protected MammifereImpl(....) // le constructeur de mammifère
        {........}
        public void allaite() {System.out.println(« J'allaite mes petits »);} // code facto
        public void reproduire() {System.out.println(« J'accouche»);} // code facto
        abstract public void mange(); // service dont on ne connait pas le code
}

Interface Herbivore extends Animal
{.............}

class HerbivoreImpl { // on ne peut la considérer comme un sous-type de Herbivore car cette classe
                        // ne pourrait être instanciable, or elle doit l'être.
        Public void mange(){System.out.println(« Je mange de l'herbe »);)
}

public class Vache extends MammifereImpl implements Herbivore
{
        protected HerbivoreImpl delegHerbivore;
        public Vache() {
        super(.....); // appel au constructeur de MamifereImpl
        delegHerbivore = new HerbivoreImpl();
        }

        public mange(){
        delegHerbivore.mange();
        }
}
```

La classe Vache est bien une classe instanciable car elle donne du code à l'ensemble des services définis par son interface fonctionnelle publique. Le code de reproduire et de allaite vient de la propriété de l'héritage de code de la classe MamifereImpl. Le code de mange déclaré abstrait dans MamifereImpl est maintenant écrit. Il se fait par délégation en utilisant le code de HerbivoreImpl.