Héritage de type

Ce document n'est pas un polycopié, c'est seulement un résumé des principales notions abordées en cours, il vient en complément du cours d'amphi et ne contient pas toutes les notions et les illustrations qui ont été présentées.

Objets, classes, interfaces.

Une **interface** définit un type en déclarant seulement l'interface fonctionnelle publiques des services d'instance. Elle ne donne pas de code au service qu'elle déclare. Le seul code qui peut-être présent dans une interface est la déclaration de constante en utilisant les mots final static. Par exemple,

```
public interface Point2D
{
        public static final PI = 3.14159;
        public void homothetie(double rapport);
        public void rotation(double angle);
        public void translation(Point2D debut, Point2D fin);
        .......
}
```

L'interface peut-être considérée comme l'implémentation d'un type abstrait de donnée. Elle ne fait que décrire un ensemble de services indépendamment de toute implémentation.

Pour définir une interface, il faut se poser la question de quelles sont les entités présentent dans le domaine du problème. Donner un nom à une interface ou à un type et la première étape qui permet d'obtenir une modalité correcte. Une fois que l'entité peut-être nommée, il est nécessaire de trouver l'ensemble des services que peut rendre ce type. A ce stade aucune considération d'implémentation ne doit être prise en compte, mais il faut que l'ensemble des services publics, soit complet et aussi minimal. Par exemple, si on considère l'interface Point2D, on ne peut pas supposer que l'implémentation de cette interface se fera en utilisant les coordonnées cartésiennes ou les coordonnées polaires. Par contre, cette interface n'est pas complète car on ne peut obtenir les coordonnées d'un point. Il faut donc la compléter en rajoutant les services publics suivants:

```
public double getX();
public double getY();
public double getRho();
```

```
public double getTheta();
```

L'héritage de type.

Une interface définit un type, l'héritage de type est une relation entre un type et un ensemble de sous type. La relation d'héritage de type se traduit comme une relation « est une sorte de » ou bien encore comme une relation « est un ». Elle établit une hiérarchie entre les types, si un Mammifère est une sorte d'animal alors le type Mammifère est un sous type du type Animal.

Par exemple,

```
public interface Vehicule
{
        public Point2D déplace(Point2D départ);
 }
```

L'interface Vehicule déclare un nouveau type « Véhicule » ainsi que le service déplace. On peut dire qu'un véhicule à la possibilité de se déplacer en allant d'un point à un autre.

Maintenant si je considère l'interface suivante:

```
public interface VehiculeTerrestre
{
        public void roule();
        public void freine();
        public voif demarre();
}
```

Cette déclaration crée un nouveau type qui est le type VéhiculeTerrestre avec comme services: roule, freine,et demarre. On peut se poser la question de savoir si il n'existe pas une relation entre le type Vehicule et le type VehiculeTerrestre.

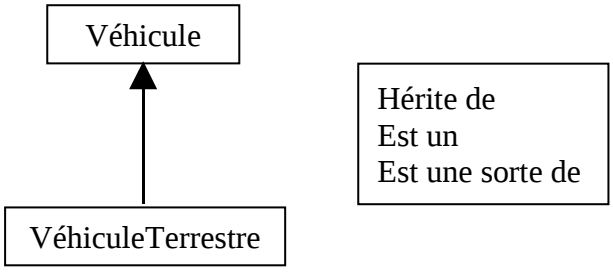
Par exemple, est ce qu'un véhiculeTerrestre n'est pas une sorte de véhicule? Est ce qu'un véhiculeTerrestre n'est pas une spécialisation d'un véhicule.

Dans ce cas, on peut considérer que le Véhicule Terrestre n'est qu'un véhicule particulier, qui se déplace sur la terre ferme et dont le déplacement peut se faire en utilisant les services demarre, freine et roule. Si on considère cette possibilité, comme un véhiculeTerrestre est un véhicule alors il doit fournir les mêmes services qu'un véhicule, c'est à dire qu'il doit se déplacer.

En java, l'héritage de type entre interface se fait en utilisant le mot clef extends. Une interface peut–être n sous type de plusieurs autres types (interfaces). Exemple de déclaration,

```
public interface VehiculeTerrestre extends Véhicule
{.....}
```

Le diagramme suivant est associé à l'héritage de type entre interface.



Les propriétés de l'héritage de type.

Si une type A hérite d'un type B, alors on a les deux propriétés suivantes:

1. **Principe de substitution :** On peut adresser les objets du sous type en utilisant le sur-type. Par exemple,

   public interface A extends B

   Alors on peut écrire A a = ... et B b = a; dans ce cas on adresse un objet de type A en utilisant une référence de type B. **On dira alors que le type A est compatible avec le type B.**

2. **Interface fonctionnelle publique de A contient tous les services de l'interface fonctionnelle de B .** IFP(B) est incluse dans IFP(A). Tout ce que peut faire un objet de type B, un objet de type peut le faire et il peut aussi en faire plus ou le faire différemment. En reprenant l'exemple précédent, un véhicule peut se deplacer donc un véhicule terrestre peut aussi se déplacer même si le service deplace n'est pas explicitement déclaré dans l'interface véhicule.

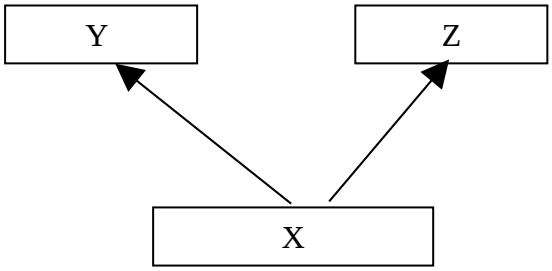
   ```
   VehiculeTerrestre vt = new .....(....); // instanciation d'une classe.
   vt.demarrer(); // service décrit dans véhicule terrestre.
   Vehicule v = vt; // première propriété (substitution);
   v.deplace(); // service déclaré dans véhicule;
   vt.deplace(); // utilisation de la seconde propriété de l'héritage.
   ```

   Sur cet exemple, v et vt sont des variables qui référencent le même objet physique, comme un véhicule terrestre est un véhicule, il est normal qu'il puisse faire la même chose qu'un Véhicule. **IL N'Y A PAS D'HERITAGE EN RESTRICTION DES SERVICES OU DE LA STRUCTURE**

Héritage Multiple.

Une interface peut-être un sous-type de plusieurs interfaces. On parle alors d'héritage multiple de type. La syntaxe est alors la suivante:

public interface X extends Y,Z.

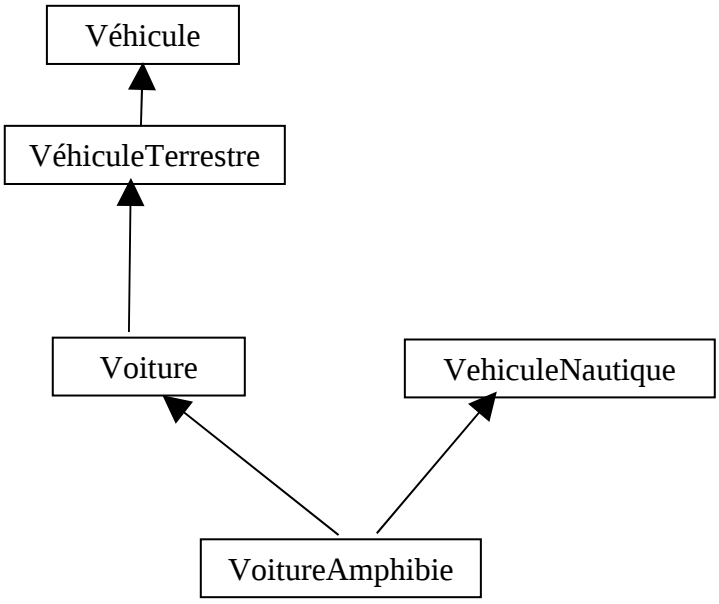


Dans ce cas, le type X est un sous type de Y et aussi un sous type de Z. On peut donc considérer à un moment un même objet physique comme étant un objet de type Y et à un autre moment considéré le même objet physique comme étant de type Z.

```
X x = new.... (); // instanciation d'une classe
Y y = x; // on considère x comme étant un Y
Z z  = x; // on considère le même objet mais comme un Z.
```

Un autre exemple:

```
public interface VehiculeNautique extends Vehicule{
        public void navigue();
}
public interface Voiture extends VehiculeTerrestre{
        public void changerRoue()
}
public interface VoitureAmphibie extends Voiture, VehiculeNautique{}
```



Dans ce cas, la Voitureamphibie cumule le comportement de VehiculeNautique, de Voiture, de VehiculeTerrestre et aussi celui de Vehicule. On peut donc, la faire rouler, freine, demarrer, naviguer et changer sa Roue. On peut aussi la convertir en tout ces types pour la manipuler.

La relation d'héritage est une relation transitive, si A hérite de B et si B hérite de C alors A hérite de C. C'est une relation anti-symetrique si A hérite de B alors B ne peut pas hériter de A. La relation d'héritage est donc une relation d'ordre partiel.

Relation entre classes et interface.

Rappel: une classe définit un type, on parlera alors de type concret car la classe permet de créer des objets, ce n'est pas le cas d'une interface qui elle ne fait que définir une interface fonctionnelle publique.

Une classe peut être considérée comme l'implémentation d'un type abstrait, c'est à dire qu'elle a en charge de fournir le code nécessaire à l'implémentation des services présentés dans l'interface fonctionnelle publique. Que cette interface soit celle de la classe elle-même ou celle d'une interface.

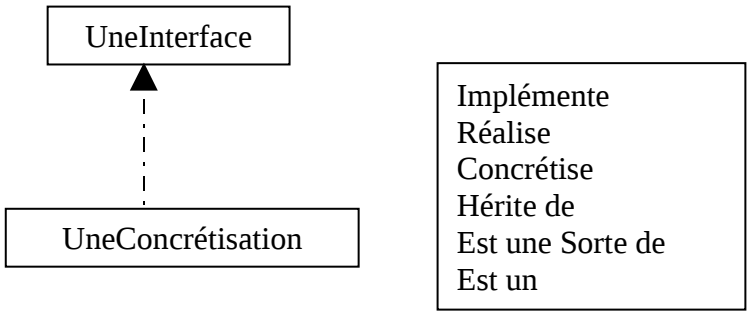
En Java, une classe peut implémenter, concrétiser, réaliser une interface. Par exemple, on peut définir l'interface du type abstrait Point2D.

```
public interface Point2D {
        public void homothétie(double rapport);
        public void translation(Point2D p, Point2D q);
        public  void rotation(double angle);
        public double getX();
        public double getY();
        public double getRho();
        public double getTheta();
}
```

Le type Point2D est un type abstrait car il est indépendant de tout choix d'implémentation. En se répétant, il ne fait que présenter un ensemble de service que peut rendre ce type.

Maintenant, il faut pouvoir concrétiser le type Point2D en faisant un choix d'implémentation pour réaliser ce type. Par exemple, on peut considérer les coordonnées cartésiennes pour représenter un point en dimension 2. La relation de réalisation entre une interface et la classe qui la réalise se fait en utilisant le mot clef **implements. La relation de réalisation entre une classe et les interfaces qu'elle concrétise est une relation type/sous type.**

```
public interface UneInterface{…….}
public class UneConcretisation {…….}
```




```
public class Point2DCartesien implements Point2D
{
        private double abs;
        private double ord;
        public void translation(Point2D p, Point2D q)
        {
          this.abs += q.getX() - p.getX();
           this.ord += q.getY() - p.getY();
        }
        // il faut ajouter le code de tous les autres services présents dans l'interface fonctionnelle.
}
```

Pour qu'une classe puisse être instanciée en utilisant new, il faut que l'ensemble des services de l'interface fonctionnelle publique soit IMPLEMENTE ; c'est à dire TOUS.

Les variables d'instances représentent la structure des objets instances de la classe. Par exemple, tous les objets dont le type réel est Point2DCartesien, auront tous les champs abs et ord. Ces champs appartiennent uniquement à l'objet, il faut donc utiliser l'objet pour pouvoir y accéder.

Une autre réalisation de l'interface Point2D aurait pu être faite par la classe Point2DPolaire qui serait déclarée de la manière suivante:

```
public class Point2DPolaire implements Point2D
{
        private double theta;
        private double rho;.....
}
```

Ainsi, le type abstrait Point2D a deux réalisations différentes qui sont donnés par les types concrets

Point2DPolaire et Point2DCartesien.

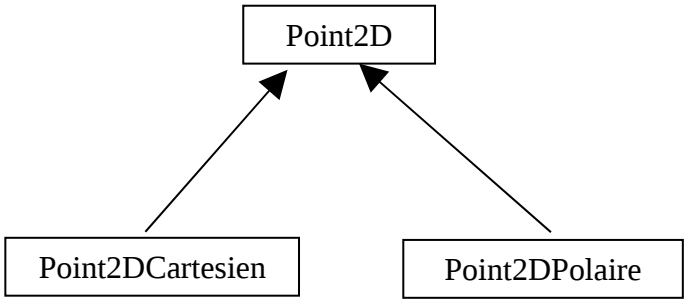
En java, il n'est pas obligatoire qu'une classe réalise une interface, en effet la déclaration d'une classe en Java, suffit pour déclarer à la fois un type abstrait et un type concret. Dans ce cas, le type abstrait et le type concret sont confondus. Pour obtenir un objet, il faut utiliser une classe.

En résumé,

- **Une interface représente un type abstrait de donnée,**
- **Une classe est la concrétisation d'un type abstrait de donnée on parle alors de type concret.**
- **Un objet est instance d'une unique classe, le type réel d'un objet est le type de la classe qui a instancié l'objet.**
- **Tous les objets qui ont le même type réel ont la même structure et le même comportement (ensemble des services de l'interface fonctionnelle publique), c'est à dire la même implémentation des services.**
- **Une classe est donc une fabrique d'objet qui définit la structure et le comportement de ces instances.**
- **Sans faire la distinction entre type réel et type concret, un type est un ensemble d'objets qui ont une structure commune et une interface fonctionnelle commune.**
- **Une interface peut-être un sous type d'une ou plusieurs interfaces.**
- **Une classe peut-être la réalisation d'une ou plusieurs interfaces, et elle ne peut hériter du code que d'une seule classe. La relation de réalisation implique la relation d'héritage de type.**
- **En java, il y a de l'héritage multiple de type et de l'héritage simple de code.**

Interface et factorisation de code.

Une interface permet de définir un type et son comportement sans supposer une quelconque implémentation, il est donc possible qu'une même interface est plusieurs réalisations. C'est par exemple, le cas de Point2D qui est une interface et qui a par exemple deux réalisations, Point2DCartesien et Point2DPolaire.



Soit par exemple, la classe Client

```
public class Client {
        static public void traiter(Point2D [] t)
        {
                for(int i = 0; i < t.length; i++)
                        t[i].homothetie(r);
        }
}
```

Cette classe ne travaille que sur le type abstrait Point2D, elle ne connait pas les concrétisations de ce type. Elle peut donc travailler avec un ensemble de Point2D sans connaître l'implémentation. Il est donc possible d'écrire le code suivant.

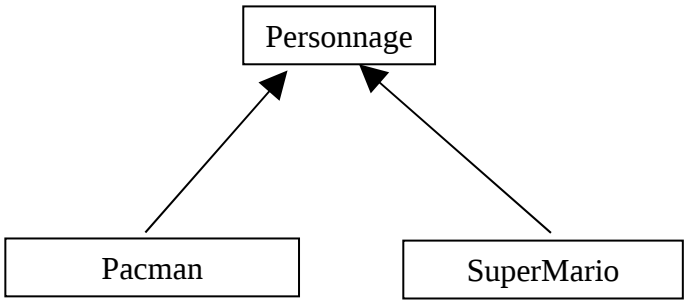
```
public class App1{
        static public void main(String [] arg) {
            Point2D [] tab = new Point2D[10];
                // on crée simplement un tableau de 10 références à des Point2D, mais il y a aucune
                instanciation de Point2D (qui est non instanciable)
                for(int i = 0; i < 5; i++)
                        tab[i] = new Point2DPolaire(i,j);
                // cette écriture est possible car Point2DPolaire est un sous type de Point2D,
                // c'est la première propriété de l'héritage de type (principe de substitution). On a
                // donc dans tab des objets dont le type déclaré est Point2D et le type réel est  //
                Point2DPolaire.

                for(int i = 5; i < 10; i++)
                tab[i] =  new Point2DCartesien(i,j);
                // cette écriture est possible car Poin2DCartesien est un sous type de Point2D.
                Client.traiter(tab); // maintenant on peut traiter un tableau qui contient à la fois des
                                        // Point2DCartesien et des Point2DPolaire.
}
```

L'utilisation de l'héritage de type permet donc d'étendre le champ d'une application sans avoir besoin de modifier le client. Sur cet exemple, si une troisième concrétisation de Point2D était fournie, on n'aurait pas besoin de reprendre le Code de client et il pourrait donc fournir des services pour ce nouveau type.

La factorisation de code qui est fournit par l'utilisation de l'héritage de type, concerne les clients de la hiérarchie de type. Nous verrons plus tard que l'utilisation de l'héritage de code permet de fournir une factorisation du code à l'intérieur de la hiérarchie de classe.

Un autre exemple est donné par le schéma suivant.

Nous pouvons considérer que le jeu ne connait que les personnages et ne dépend pas des personnages concrets qui sont Pacman et SuperMario.



Le code suivant :

```
public class Jeu
{
        public void anime(Personnage p)
        {
                p.deplace();
                p.anime();
        }
}
```

Pour que ce mécanisme fonctionne, il faut que l'interface personnage contienne le comportement commun à Pacman et à SuperMario. En utilisant la propriété 2 de l'héritage de type, nous avons IFP(Personnage) qui est incluse dans l'intersection de IFP(Pacman) et de IFP(SuperMario).

Mais cela n'est pas suffisant, il faut que les Pacman et les SuperMario conservent leur comportement quand ils sont vus comme des personnages. Par exemple, si on demande à un personnage de se dessiner, il faut que ce personnage se dessine comme un Pacman, si il a été créé par la classe Pacman, ou comme un SuperMario si il a été créé par la classe SuperMario. Il faut donc que la propriété suivante soit vrai.

Un objet doit conserver le comportement qui lui a été donné au moment de son instanciation, indépendamment du type qui le référence. Un pacman doit se comporter comme un pacman tout au long de sa vie, même si on ne le perçoit comme un personnage.

Héritage de Type et décomposition ensembliste.

Nous avons vu précédemment comme un type peut être associé à un ensemble d'objet ayant un comportement commun, si on passe au niveau des classes on doit aussi prendre en compte la structure des objets. Comme la relation d'héritage de type, se traduit par « est un » ou « est une sorte de ».

La relation type sous-type peut se traduire sous la forme d'inclusion d'ensemble. Si A est un sous type de B, alors l'ensemble des objets A est inclus dans l'ensemble des objets B. On dira que A est un sous ensemble de B. Si A est un sous type de B et de C (héritage multiple), alors A est un sous ensemble de B et A est aussi un sous ensemble de C, donc A est un sous ensemble de B inter C. L'héritage Multiple est une cumulation des comportements des sur types et une intersection d'ensemble.

Prenons par exemple, les spécifications suivantes:

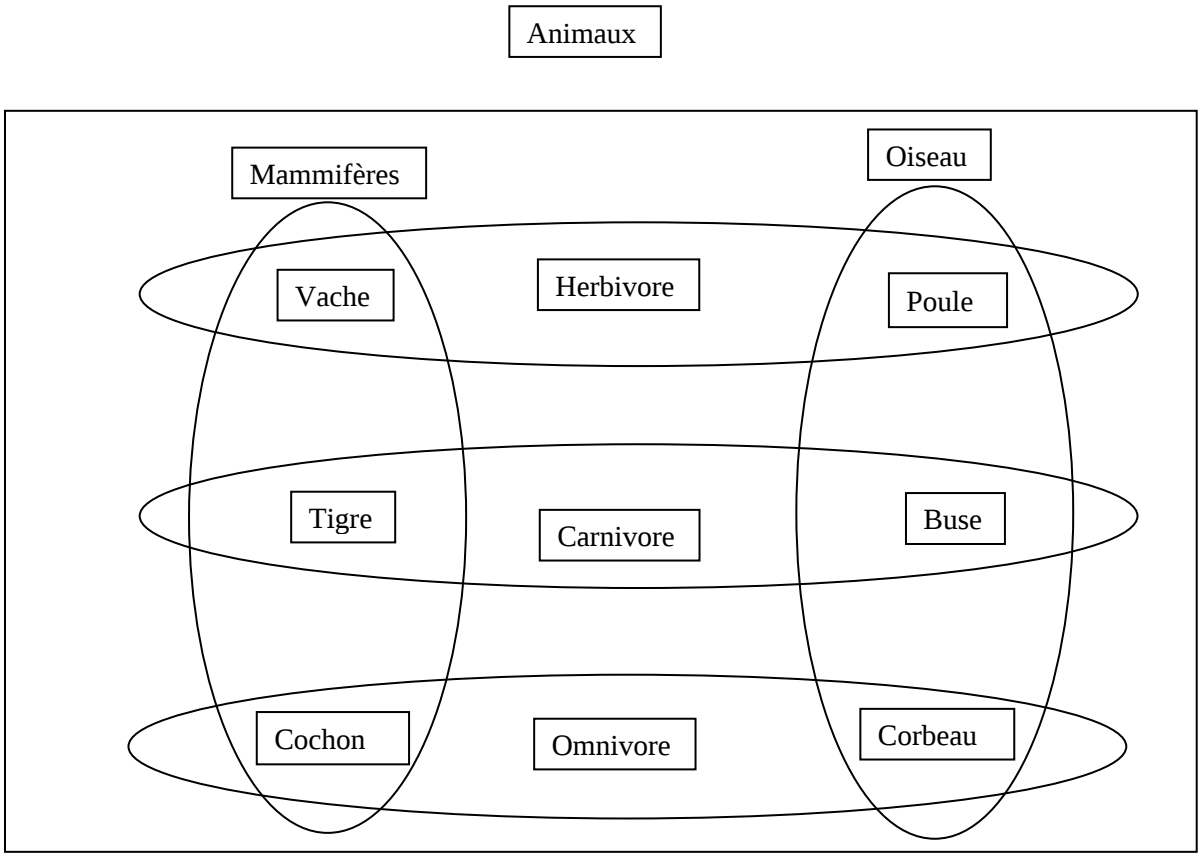
Les oiseaux mangent et se reproduisent. Les mamifères sont des animaux qui allaitent et qui se reproduisent en accouchant. Les oiseaux sont des animaux qui volent et qui se reproduisent en pondant des oeufs. Parmi, les animaux les herbivores mangent de l'herbe, les carnivores mangent de la la viande, les omnivores ne sont ni des herbivores ni des des carnivores mais ils mangent parfois comme un herbivore et parfois comme un carnivore. Les poules sont des oiseaux herbivores, les vaches sont des mamiferes herbivore, les chauves souris sont des mamifères omnivores qui volent, le tigre est un mamifere carnivore, le corbeau est un oiseau qui mange de la viande et de l'herbe.

On ne s'intéresse pour l'instant qu'à la relation type sous-type. C'est à dire exclusivement à l'héritage de type et non à l'héritage de code.

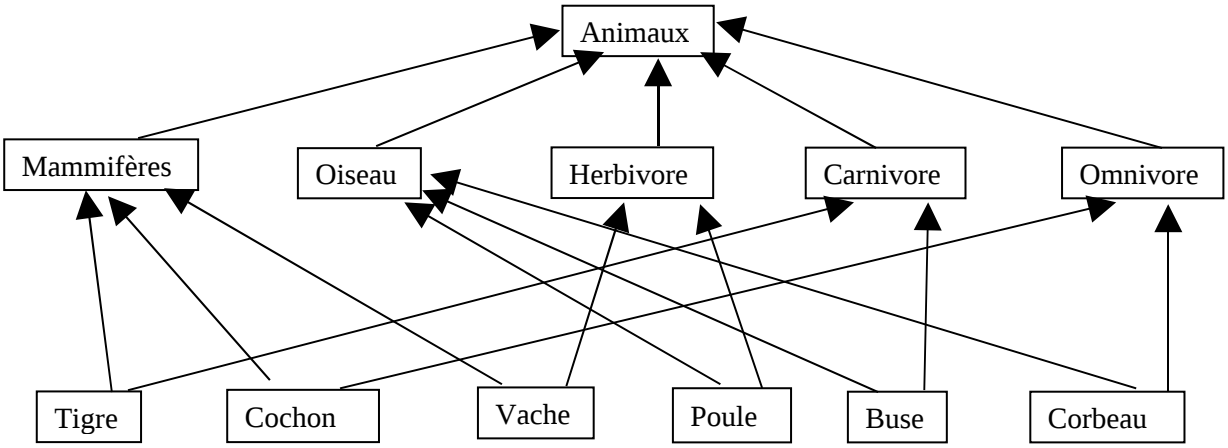
On suppose qu'il n'existe pas d'animaux qui sont à la fois des mammifères et des oiseaux. L'intersection entre les oiseaux et les mammifères est égal à l'ensemble vide. Donc, en aucun cas la chauve souris qui est un mammifère ne peut-être un sous-type de oiseaux même si la chauve souris vole.

Il en est de même pour les herbivores et les carnivores, même si les omnivores mangent comme les herbivores et les carnivores, cette particularité concerne le code et non les types. L'intersection entre les carnivores et les omnivores est elle aussi égale à l'ensemble vide.

A partir des spécifications précédentes, on obtient donc le diagramme ensembliste suivant:



Ce diagramme ensembliste se traduit donc sous la forme suivante en terme de hiérarchie de type.



Maintenant nous devons nous intéresser à définir les interfaces et les classes de notre application.

Concernant le type Animal, à ce stade nous n'avons aucune information sur comment les animaux mangent et se reproduisent. Nous ne connaissons donc que l'interface fonctionnelle de Animal.

```
public interface Animal{
        public void mange();
        public void reproduire();
}
```

Nous savons que les Mammifères sont des Animaux qui ont une particularité supplémentaire celle d'allaiter. Il s'agit donc d'une extension du comportement de Animal. Nous savons également comment il se reproduisent, c'est à dire que nous pourrions donner du code aux services allaite et reproduire. Par contre, nous ne pouvons pas donner du code pour le service mange. Donc, si nous faisions de Mammifère une classe, elle ne pourrait être instanciable car elle ne fournit pas une implémentation pour l'ensemble des services décrits par l'interface fonctionnelle publique. Nous allons donc en faire une interface.

```
public interface Mammifère extends Animal
{
        public void mange();
        public void reproduire(); //code possible
        public void allaite(); // extension du comportement de Animal.
}
```

Nous avons exactement les mêmes caractéristique pour le type Oiseau, avec une extension du comportement avec le service vole, et la possibilité de définir du code pour reproduire, mais nous n'avons toujours pas la possibilité de donner du code au service mange. Nous avons donc une nouvelle interface qui est la suivante.

```
public interface Oiseau extends Animal
```

```
{
        public void mange();
        public void reproduire(); // code possible
        public void vole(); // extension du comportement de Animal.
}
```

Pour les types Herbivore, Carnivore Omnivore, il n'y a pas d'extension du comportement. On peut donner du code pour le service mange, mais comme Herbivore, Carnivore et Omnivore sont des sous-types de Animal, ils doivent aussi posséder du code pour le service reproduire. Mais ceci, n'est pas possible donc pour les mêmes raisons que précédemment, à notre niveau de connaissance, ces trois types ne peuvent être représentés par des classes instanciables, ils seront donc associés a des interfaces. Nous ne montrerons que la déclaration de l'interface Herbivore, les autres seront identiques.

```
   Public interface Herbivore extends Animal
{
        public void mange();  // code possible
        public void reproduire();
}
```

Maintenant, il nous reste a voir toutes les autres types, qui sont Cochon, Vache, Tigre, Poule, Chauve souris et Corbeau. Pour tous ces types, on peut donner du code pour tout leur service public. Mais nous allons voir que nous allons créer de la duplication de code. En effet, si l'on considére par exemple que tous les herbivores mangent de la même manière, le code pour faire manger la poule et la vache se trouveront déclarés dans ces deux classes. Si l'on considère que tous les mamiféres allaitent de la même manière, il y aura de la duplication de code pour le service allaite de tous les mamifères. Par contre, comme pour tous ces types on peut donner une implémentation de tous les services, on peut maintenant les concrétiser en faisant une classse. Par exemple,

```
public class Cochon implements Omivore, Mamifères // heritage multiple de type
{
        public void allaite (){ // code dupliquer pour tous les mamifères
                System.out.println(« Allaite »);
        }
        public void reproduire(){ // code dupliquer pour tous les mamifères
                System.out.println(« Reproduire »);
        }
        public void mange(){ // code dupliquer pour tous les Omnivores
                System.out.println(« mange  comme un herbivore »); // code de Herbivore
                System.out.printl(«  mange comme un carnivore »); // code de carnivore.
        }
}
```

Un autre exemple avec la chauve souris qui vole comme un oiseau mais qui n'est pas un oiseau.

```
public class ChauveSouris implements Omivore, Mamifères // heritage multiple de type mais aucun
                                                          //  rapport avec le type oiseau
{
        public void vole (){ // code dupliquer si la chauve souris vole comme les oiseaux
                System.out.println(« vole»);
        }
        public void allaite (){ // code dupliquer pour tous les mamifères
                System.out.println(« Allaite »);
        }
        public void reproduire(){ // code dupliquer pour tous les mamifères
                System.out.println(« Reproduire »);
        }
        public void mange(){ // code dupliquer pour tous les Omnivores
                System.out.println(« mange  comme un herbivore »); // code de Herbivore
                System.out.printl(«  mange comme un carnivore »); // code de carnivore.
        }
}
```

Nous verrons par la suite comment factoriser le code pour éviter la duplication au sein d'une application. Par cela, il faudra utiliser les notions de classe abstraite et d'héritage de code.