

Remise à niveau Programmation

Programmation Objet avec Java

**Variable de classe, Méthodes de classe
et autres constantes**

Héritage

Hiérarchie de classe

Polymorphisme

#3

Licence Pro
Métiers de l'Informatique : Conception, Développement,
Test de Logiciels, Parcours "Génie Logiciel, Système d'Information"
2020-2021

Constantes - final

Les constantes sont des variables non modifiables. Par convention le nom des variables constantes est écrit en **MAJUSCULE**.

Pour indiquer qu'une variable est une constante on ajoute le modificateur **final** à la déclaration de la variable.

Exemple :

```
{  
    public final double ALTITUDE_MAX = 20000. ;  
}
```

Le compilateur rejettera toute instruction cherchant à affecter une nouvelle valeur à la constante.

Remarque

La première affectation (celle qui détermine définitivement la valeur de la constante) peut être donnée tardivement (après la déclaration) .

```
{  
    public final double ALTITUDE_MAX ;  
    ALTITUDE_MAX = 3 ;  
}
```

Il est donc tout à fait possible de définir une variable d'instance comme étant constante et de confier cette première affectation à un constructeur :

```
public class PointPlan  
{  
    private double abscisse;  
    private double ordonnée;  
    private final String NOM ;    // ----- Déclaration de la constante  
  
    public PointPlan(double x, double y, String n)  
    {  
        this.setAbscisse( x ) ;  
        this.setOrdonnée( y ) ;  
        this.NOM = n ;           // ----- Définition de la valeur  
                                // ----- de la constante  
    }  
}
```

Ainsi, chaque instance de **PointPlan** possède un nom qui est une constante. Une fois le nom donné par le constructeur, celui-ci ne peut plus être modifié.

Variable référence Constante

Il est possible de définir une variable référence comme constante en utilisant le modificateur **final**.

```
{
    final PointPlan ORIGINE ;
    ORIGINE = new PointPlan( 0., 0., "O " ) ;
}
```

Comme pour les types primitifs une fois la référence affectée à la variable, celle-ci ne peut plus être modifiée. Ainsi l'instruction rajoutée dans le code suivant (la dernière instruction) n'est pas acceptée par le compilateur :

```
1 {
2   final PointPlan ORIGINE ;
3   ORIGINE = new PointPlan( 0., 0., "O " ) ;    // ----- Acceptée
4   ORIGINE = new PointPlan( 10., 20., "O " ) ; // ----- Rejetée
5 }
```

La référence contenue dans **ORIGINE** est une constante, or ici on tente de modifier la référence contenue dans la constante **ORIGINE** après sa définition de la ligne 3 (affectation à la ligne 4 de la référence d'un nouveau point renvoyé par l'opérateur **new**). Ce n'est pas permis !

Attention

Ce n'est pas parce que la variable-référence est constante que l'objet référencé l'est !

Ici, l'objet point plan référencé peut tout a fait être modifié. Ainsi le code suivant est parfaitement valide :

```
{
    final PointPlan ORIGINE ;
    ORIGINE = new PointPlan( 0., 0., "O " ) ;
    ORIGINE.setAbscisse( 10.) ; // -- La référence n'est pas modifiée
    ORIGINE.setOrdonnée( 20.) ; // -- La référence n'est pas modifiée
}
```

Schéma :



Variable de classe - static

Une variable de classe est associée à une classe et non à une instance particulière (objet). Elle décrit donc une propriété de la classe et non d'une instance.

En corolaire, on peut dire qu'une variable de classe n'existe qu'en 1 seul exemplaire alors qu'une variable d'instance possède autant d'exemplaires qu'il y a d'instances.

Une variable de classe est déclarée en utilisant le modificateur **static**

Exemple : Maintenir un compteur d'instance

On peut mettre en place un compteur qui est incrémenté à chaque fois qu'une nouvelle instance est créée.


```
public class PointPlan
{
    private double abscisse ;
    private double ordonnée;

    public static int nbPointCréés = 0 ;    // ----- Déclaration

    public PointPlan( double x, double y)
    {
        this.setAbscisse(x) ;
        this.setOrdonnée(y) ;

        PointPlan.nbPointCréés++ ;        // ----- Appel
    }

    public PointPlan()
    {
        this(0., 0.);                      // ----- Appel indirect
    }
}
```



Remarque

L'accès à la variable de classe ne se fait pas au moyen du mot clef **this**. Il se fait en faisant précéder le nom de la variable de classe par le nom de la classe ici **PointPlan**.

```
public static void main( String [] args)
{
    PointPlan p1 = new PointPlan() ;
    PointPlan p2 = new PointPlan( 3., 12.) ;

    System.out.println( PointPlan.nbPointCréé) ; // ----- Appel
}
```

Variable de classe - static

Schéma :



Comme la variable de classe n'existe qu'en un seul exemplaire et qu'elle n'est rattachée à aucune instance particulière mais directement invoquée au moyen du nom de la classe, alors :

Il est possible d'utiliser la variable de classe sans avoir instancié aucun objet de cette classe.

Le code suivant est donc parfaitement valide.

```
public static void main( String [] args)
{
    System.out.println( PointPlan.nbPointCréé) ; // ----- Appel
}
```

Il affiche 0.

Remarque

Il est possible d'utiliser tous les modificateurs sur une variable de classe (donc en plus du modificateur **static**).

- **public static** ... : variable de classe publique
- **private static** ... : variable de classe privée
- **public static final** ... : constante de classe publique
- **private static final** ... : constante de classe privée

Méthode de classe - static

Une méthode de classe s'applique à une classe et non à un objet.


Une méthode de classe décrit un service rendu par la classe, service qui n'est lié à aucun objet.

Comme pour les variables de classe, la déclaration d'une méthode de classe se fait au moyen du modificateur **static**.

```
public class PointPlan
{
    private double abscisse ;
    private double ordonnée;

    private static int nbPointCréés = 0 ; // Variable de classe

    public static int getNbPointCréés() // Méthode de classe publique
    {
        return PointPlan.nbPointCréés ;
    }
}
```



Remarque


L'accès à la méthode de classe ne se fait pas au moyen d'une variable référence. Elle se fait en faisant précéder le nom de la méthode de classe par le nom de la classe ici **PointPlan**.

L'appel suivant est donc valide :

```
public static void main( String [] args)
{
    System.out.println(PointPlan.getNbPointCréés());

    PointPlan p1 = new PointPlan() ;
    PointPlan p2 = new PointPlan( 3., 12.)

    System.out.println(PointPlan.getNbPointCréés());
}
```



Affiche **0** puis **2**;

Remarque

Le mot clef **this**, ne peut apparaître dans une méthode de classe puisque celle-ci ne dépend d'aucune instance à laquelle ferait référence **this**.

public static void main(...)

Une méthode de classe particulière

`main()` est une méthode de classe particulière. C'est la seule à pouvoir être invoquée par le système d'exploitation.

Soit le code suivant :


```
public class TestPointPlan
{
    public static void main( String [] args)
    {
        ...;
    }
}
```

Lorsqu'on appelle la commande `java TestPointPlan` cela revient à invoquer l'appel de la méthode de classe `main` de la classe `TestPointPlan`

Passer des valeurs au programme depuis le système

```
public class TestPointPlan
{
    public static void main( String [] args)
    {
        double x ;
        double y ;
        PointPlan p;

        if ( args.length == 2 )
        {
            // Double.parseDouble( f ) convertis en double la chaine f
            x = Double.parseDouble( args[0] ) ;
            y = Double.parseDouble( args[1] ) ;
            p = new PointPlan( x, y ) ;
        }
    }
}
```



L'appel suivant crée donc un `PointPlan` avec les coordonnées `3.` et `7.4` passées en paramètre.

```
login@bash> java TestPointPlan 3 7.4
```



Schéma :



Héritage

Le concept d'héritage est un concept central en programmation orientée objet. C'est un principe qui permet de rendre le code modulaire en factorisant certaines propriétés et/ou comportements dans une classe et de propager ces propriétés ou ces comportements à d'autres classes grâce à un mécanisme appelé *héritage*.

Exemple de code redondant :

```
public class Animal {
    private double poids ;

    public Animal( double p){
        this.setPoids( p) ;
    }

    public void setPoids(double p){
        this.poids = p ;
    }

    public double getPoids(){
        return this.poids ;
    }

    public void affiche(){
        System.out.println(this.poids) ;
    }
}
```

```
public class Mammifère {
    private double poids ;
    private int nbMamelles ;

    public Mammifère( double p, int m){
        this.setPoids( p) ;
        this.setNbMamelles( m) ;
    }

    public void setPoids( double p){
        this.poids = p ;
    }

    public double getPoids(){
        return this.poids ;
    }

    public void setNbMamelles(int n){
        this.nbMamelles = n ;
    }

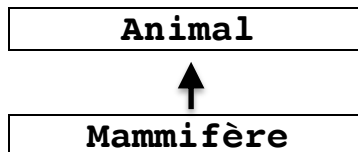
    public int getNbMamelles(){
        return this.nbMamelles ;
    }

    public void affiche(){
        System.out.println(this.poids) ;
        System.out.println(this.nbMamelles ) ;
    }
}
```

Le principe d'héritage permet d'éviter cette redondance en déclarant que « un **Mammifère** est une sorte d'**Animal**» lors de la déclaration de la classe grâce au mot-clef **extends**:

```
public class Mammifère extends Animal{...}
```

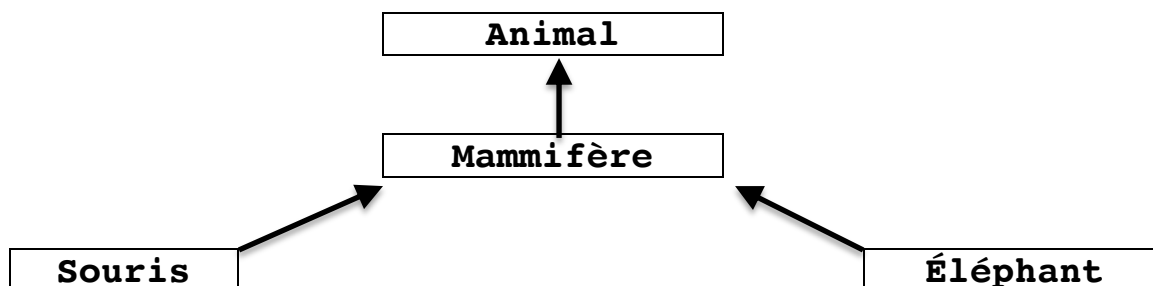
Ainsi, automatiquement et implicitement, les instances de la classe **Animal** disposent, sans avoir à répéter le code, de tous les services de la classe **Mammifère**.



- La classe **Mammifère** [*hérite, étend, est une classe fille de, est une sous-classe de, est une classe dérivée de, est plus spécifique que*] la classe **Animal**.
- La classe **Animal** [*est LA classe mère de, est -LA sur-classe de, est plus générale que*] la classe **Mammifère**

Héritage

Intérêts

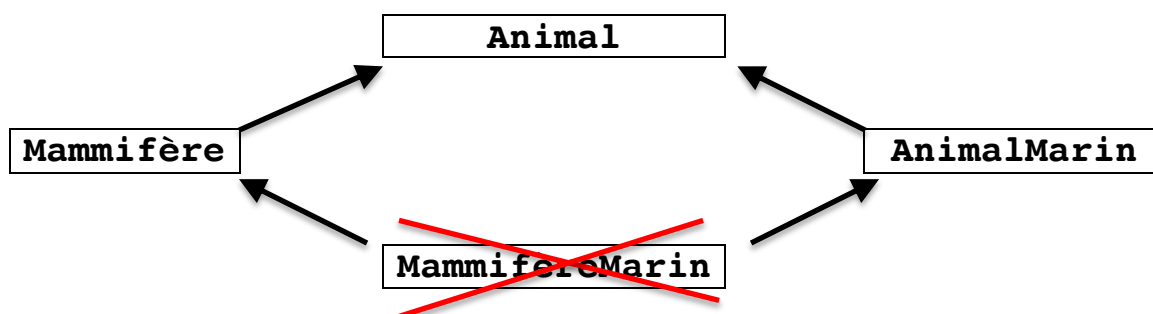


- Réutilisation : concevoir de nouvelles classes à partir de classes déjà définies
gain important de temps de développement + minimisation des risques d'erreurs.

- Factorisation : mise en commun des variables et méthodes
gain de place + lisibilité + sûreté

Remarque

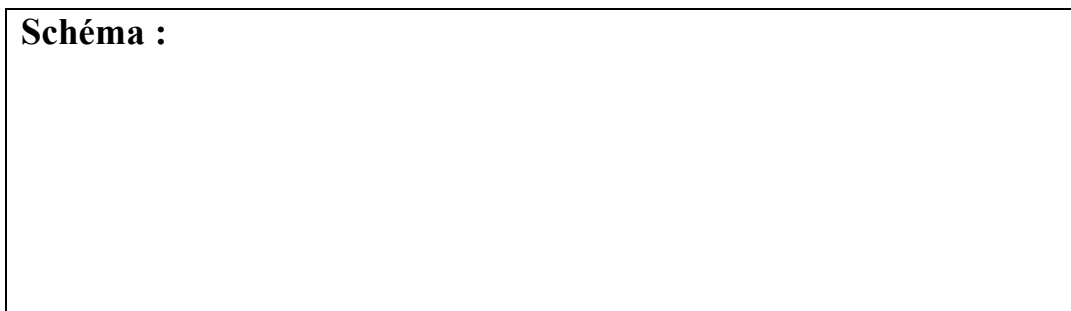
Même si UML et les principes de programmation objet l'autorise, l'héritage multiple n'est pas implémenté en Java.



Pour spécifier qu'une classe A hérite d'une classe B on utilise la syntaxe suivante :

```
public class Animal {...}
public class Mammifère extends Animal {...}
public class AnimalMarin extends Animal {...}
```

Schéma :



Objets de la classe fille

Soit la classe **Animal** précédemment définie. On définit **Mammifère** ainsi

```
public class Animal
{
    private double poids ;

    public Animal( double p){
        this.setPoids( p ) ;
    }

    public Animal(){
        this.poids = 5000.f ;
    }

    public void setPoids(double p) {
        this.poids = p ;
    }

    public double getPoids() {
        return this.poids ;
    }

    public void affiche(){
        System.out.println(this.poids) ;
    }
}

public class Mammifère extends Animal
{
    private int nbMamelles ;

    public Mammifère( double p, int m){
        super( p ) ;
        this.setNbMamelles( m ) ;
    }

    public Mammifère(int m) {
        super() ;
        this.nbMamelles = m ;
    }

    public void setNbMamelles(int n) {
        this.nbMamelles = n ;
    }

    public double getNbMamelles() {
        return this.nbMamelles ;
    }

    public void affiche(){
        super.affiche() ;
        System.out.println(this.nbMameles) ;
    }
}
```

Dans la classe fille (ici **Mammifère**).

- **super** permet d'accéder à une méthode de la classe mère. (ici **super.affiche()** est un appel à la méthode affiche de la classe **Animal**)
- **super()** permet de faire appel au constructeur de la classe mère. Si elle est utilisée cette instruction doit impérativement être la première instruction du constructeur de la classe fille

Lors de l'instanciation d'un objet de la classe fille (ici **Mammifère**), celui-ci sera défini

- avec sa variable d'instance **nbMamelles** spécifique à la classe fille **Mammifère**,
- avec sa variable d'instance **poids** héritée de la classe mère **Animal**.

L'initialisation de la valeur de la variable d'instance héritée est confiée au constructeur de la classe mère par l'appel à **super()**.

```
public class TestMammifère {
    public static void main( String [] args )
    {
        Mammifère souris = new Mammifère(0.05, 10) ;
        Mammifère éléphant = new Mammifère(2) ;
    }
}
```

Méthodes héritées

Avec l'héritage, le nombre de méthodes associées à une instance d'une classe fille peut devenir très important. Les instances de la classe fille peuvent

- invoquer les méthodes qui leur sont spécifiques (celles qui sont définies dans le corps de la définition de la classe) ;
- invoquer les méthodes dont elles héritent de leur classe mère ou d'une de leur classe parente.

```
{  
    Animal humain = new Animal (50) ;  
    Mammifère éléphant = new Mammifère(5000., 2) ;  
  
    // invocation d'une méthode spécifique à la classe Mammifère  
    int n = éléphant.getNbMamelles() ;  
  
    // invocation d'une méthode héritée de la classe Animal  
    double pE = éléphant.getPoids() ;  
  
    // invocation d'une méthode spécifique de la classe Animal  
    double pH = humain.getPoids() ;  
  
    // invocation d'une méthode spécifique de la classe Animal  
    humain.affiche() ;  
  
    // invocation d'une méthode redéfinie dans la classe Mammifère  
    éléphant.affiche() ;  
}
```

Schéma :



Accessibilité - portée

Encapsulation

Les variables privées de la classe mère sont inaccessibles aux méthodes des classes filles. L'encapsulation l'interdit.

```
public class Animal
{
    private double poids;

    ...
}
```

Alors, les instructions suivantes ne sont pas toutes valides :

```
public class Mammifère extends Animal
{
    private int nbMamelles ;

    public void affiche()
    {
        System.out.println( this.poids) ;           // invalide
        System.out.println( super.poids) ;         // invalide
        System.out.println( super.getNbMamelles() ); // invalide
        super.affiche()                             // valide
        System.out.println( this.nbMameles );       // valide
    }
}
```

Perméabilisation de l'encapsulation

Le modificateur d'accès **protected** est intermédiaire entre les modificateurs **public** et **private**.

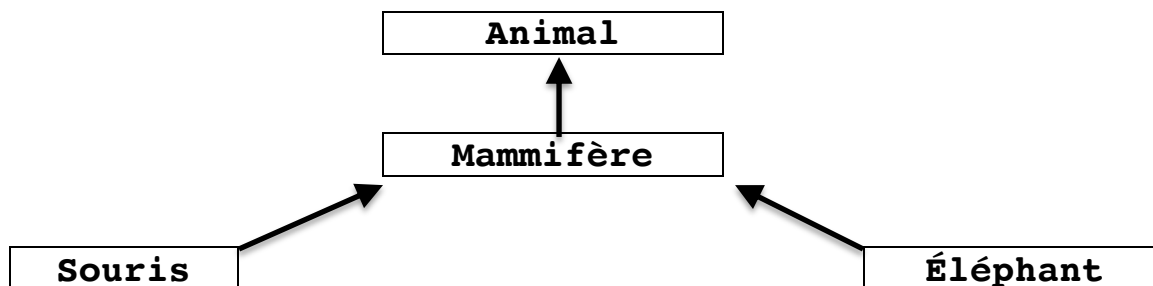
Il permet de donner accès à une variable d'instance définie dans une classe mère à partir d'un corps d'instruction exprimé dans une méthode d'une classe fille ou d'une classe du même package.

```
public class Animal
{
    protected double poids;

    ...
}
```

ON NE LE FAIT PAS. On préserve autant que possible l'encapsulation !!

Hiérarchie de classe



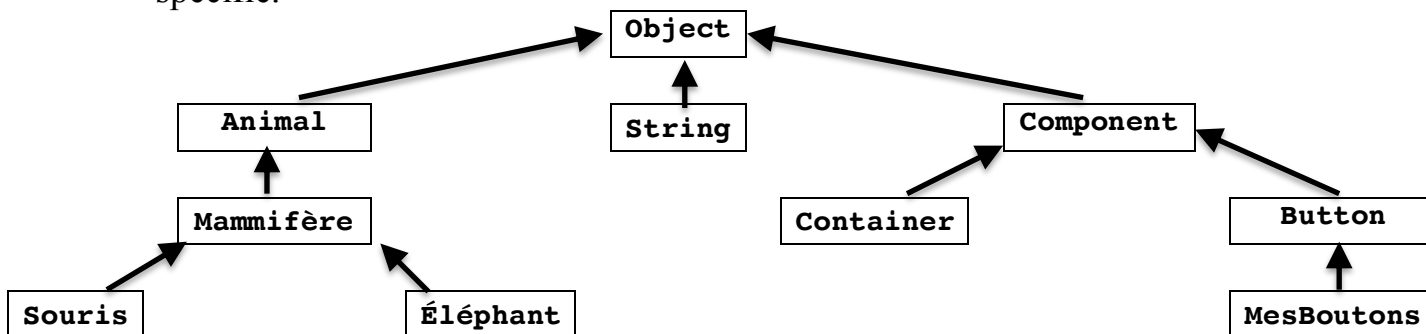
Souris hérite de **Mammifère** et **Mammifère** hérite de **Animal**.
Par transitivité **Souris** hérite de **Animal**.

Définition

- **Souris** est une classe *fil*le de **Mammifère**
- **Souris** est une classe *descendante* de **Animal**
- **Souris** a pour classe *mère* la classe **Mammifère**
- **Souris** a pour classe *ascendante* la classe de **Animal**

Remarque

Toutes les classes descendent de la classe **Object** Même si dans leur déclaration il n'en est pas fait mention. Le « **extends Object** » est implicite si rien n'est spécifié.



Remarque

- Une classe est compatible avec tous ses ascendants Un **Éléphant** est un **Mammifère** qui est lui même un **Animal**. On peut donc *utiliser* un **Éléphant** partout où on attend un **Mammifère** ou un **Animal**.
- Tous les services que l'on est en droit d'attendre d'un **Animal** peuvent être fournis par une instance d'un de ses classes descendantes, par exemple par un **Mammifère** ou un **Éléphant** qui sont plus spécifiques :

```
{  
    Animal a = new Mammifère(50, 4) ; // est parfaitement correct  
}
```

Hiérarchie de classe

```
public class Animal
{
    private double poids;

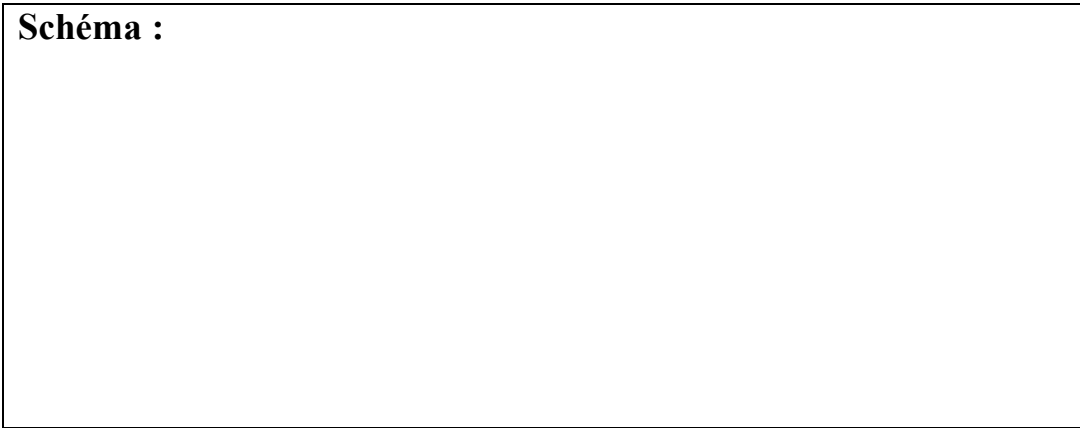
    public static void affichePoids(Animal a)
    {
        System.out.println( "Poids = ", a.getPoids() ) ;
    }
}
```

```
public class Mammifère extends Animal
{
    ...
}
```

Le code suivant est valide puisque on passe à la méthode **affichePoids()** un **Mammifère** qui est un **Animal** *par héritage*.

```
public class TestAnimal
{
    public static void main(String [] args)
    {
        Mammifère m = new Mammifère( 5000., 2 ) ;
        Animal.affichePoids( m ) ;
    }
}
```

Schéma :



Remarque

Si X est descendante de Y alors :

- on peut toujours fournir une instance de X où on attend une instance de Y.
X est une sorte de Y, X a tout ce qu'il faut comme Y.
- on ne peut pas fournir une instance de Y là où on attend une instance de X.
cas X a une spécialisation que n'a pas Y.

Polymorphismes

Signature d'une méthode

La *signature* d'une méthode est constituée

- du nom de la classe où elle est définie,
- du nom de la méthode
- des types et de l'ordre de ses paramètres.

Le type de la valeur de retour ne fait pas partie de la signature de la méthode.

Polymorphisme de Surcharge

On parle de polymorphisme de surcharge quand au moins deux méthodes de la même classe, portent le même nom. Elles diffèrent alors par le nombre et/ou le type-ordre de ses paramètres.

Dans l'exemple ci dessous

- c'est le cas des constructeurs **PointPlan()** : ils sont définis dans la même classe, il portent le même nom mais ils diffèrent par le nombre et le type de leurs arguments.

- c'est le cas des constructeurs **symétrique()** : ils sont définis dans la même classe, il portent le même nom mais ils diffèrent par le nombre et le type de leurs arguments.

```
public class PointPlan{
    private double abscisse;
    private double ordonnée;
    private final String NOM ;

    public PointPlan(double x, double y, String n)
    {
        this.setAbscisse( x) ;
        this.setOrdonnée( y) ;
        this.setNOM( n) ;
    }
    public PointPlan()
    {
        this( 0., 0., "M");
    }

    public PointPlan symétrique()
    {
        return new PointPlan( -this.getAbscisse(), -this.getOrdonnée() ) ;
    }

    public PointPlan symétrique(PointPlan p)
    {
        return new PointPlan( 2*p.getAbscisse()-this.getAbscisse(),
                               2*p.getOrdonnée()-this.getOrdonnée() ) ;
    }
}
```

Polymorphismes

Polymorphisme de redéfinition (ou d'héritage)

On parle de polymorphisme de redéfinition ou de polymorphisme d'héritage lorsque deux méthodes de deux classes liées par une relation d'héritage (mère-fille ou ascendante-descendante) portent le même nom, présentent le même nombre, type et ordre dans leurs paramètres.

Elles ont bien des signatures différentes puisqu'elles diffèrent par le nom de la classe dans laquelle elles sont définies.

```
public class Animal
{
    ...
    public void quiSuisJe()
    {
        System.out.println("Je suis un animal") ;
    }
}
```

```
public class Mammifère extends Animal
{
    ...
    public void quiSuisJe()
    {
        System.out.println("Je suis un Mammifère") ;
    }
}
```

Ici la méthode **quiSuisJe()** définie dans la classe **Animal** est redéfinie dans la classe **Mammifère**.

La classe **Mammifère** dispose donc de deux méthodes **quiSuisJe()** :

- une méthode héritée de la classe **Animal**,
- une méthode spécifique

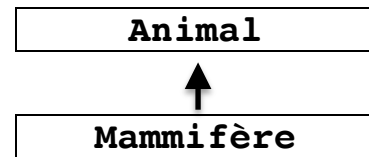
Il peut alors se poser des questions non-triviales quant à savoir quelle méthode est invoquée lors d'un appel. Par exemple, dans la situation suivante, ce n'est pas évident :

```
{
    Animal a = new Mammifère(5000., 2) ;

    a.whoAmI() ;
}
```

Liaison statique - dynamique

Soient deux classes **Animal** et **Mammifère** liées par une relation d'héritage :



Soit le code suivant :

```
{  
    Animal ref ;  
    ref = new Mammifère( 5000., 2) ;  
    ref.quisuisJe() ;  
}
```

- La variable référence **ref** est de type **Animal**,
- La référence placée dans la variable **ref** renvoie à une instance de **Mammifère** (ce qui est parfaitement possible puisque **Mammifère** est une classe fille de **Animal** et peut donc être utilisée partout où l'on attend un **Animal**).

Compilation : Liaison statique

Lors de la compilation, le compilateur vérifie l'existence d'une méthode dans la classe de la variable référence (ici **Animal**) préfixant l'appel de la méthode (ici **quisuisJe()**).

Si la classe de déclaration de la variable (ici **Animal**) n'a pas de méthode de même signature alors il y a un rejet du compilateur.

Pas de vérification sur **Mammifère** lors de la compilation !

Exécution: Liaison dynamique

Lors de l'exécution du programme, la machine virtuelle recherche la méthode sélectionnée par le compilateur dans la classe de l'objet désigné par la variable référence (ici **Mammifère**) préfixant l'appel de la méthode. Si il en trouve une il l'exécute ;

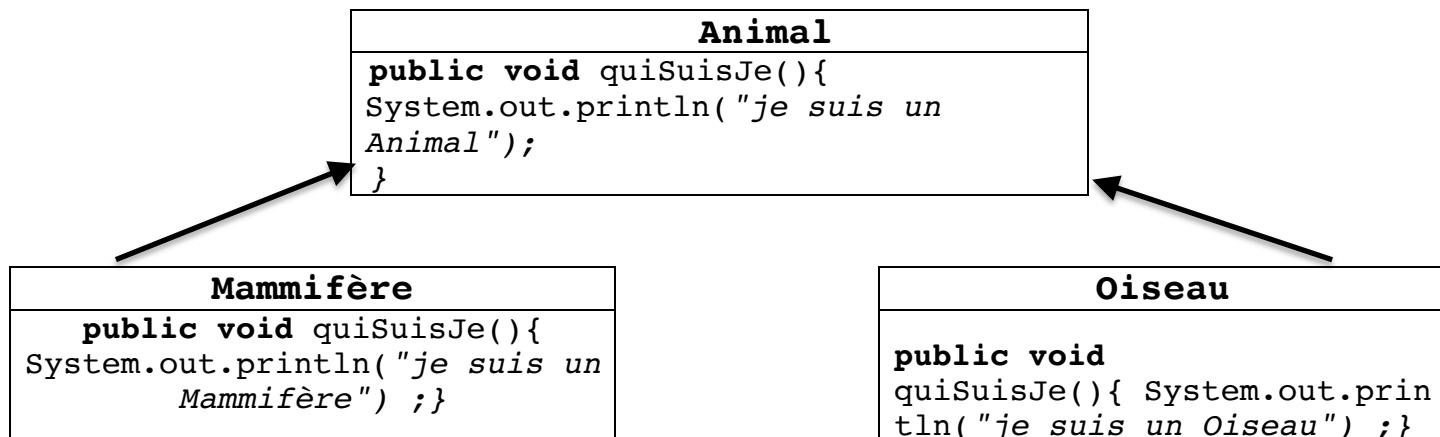
Si il n'y en a pas, la machine virtuelle remonte dans l'arborescence d'héritage (ici vers **Animal**) pour regarder si elle ne trouve pas une méthode de la même signature dans les classes ascendantes (plus générales).

La machine virtuelle exécutera toujours la première méthode de la hiérarchie de classe présentant la bonne signature (toujours la méthode la plus spécifique).

Elle en trouvera forcément une puisque le code exécuté par la machine virtuelle a passé la compilation !

Exemple de polymorphisme

Soit la hiérarchie de classe suivante :



Le polymorphisme est un outil puissant permettant une grande adaptabilité du code. Ainsi, dans le code suivant une même instruction fait appel à 3 méthodes `quiSuisJe()` différentes !

```
Public class TestPolymorphisme
{
    public static void man( String [] args )
    {
        Animal [] tab = new Animal [3] ;

        tab[0] = new Animal( 50.) ;
        tab[1] = new Mammifère( 5000., 2) ;
        tab[2] = new Oiseau(1.2, 0.8) ;

        for ( int i = 0 ; i < tab.length ; i++)
            tab[i].quiSuisJe() ;

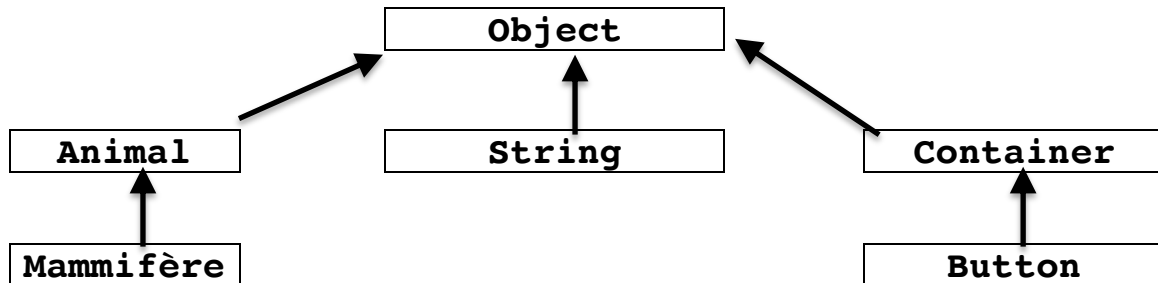
        // for ( Animal a : tab)
        //     a. quiSuisJe() ;
    }
}
```

L' exécution de ce code produit les affichages suivants :

```
Je suis un Animal
Je suis un Mammifère
Je suis un Oiseau
```

Object & Polymorphisme

Toute nouvelle classe hérite automatiquement de la classe `Object()` même si la déclaration de la classe n'y fait pas référence explicitement.



Méthodes de la classe Object()

Parmi les méthodes dont toute classe hérite de la classe `Object`, il en est 2 qu'il faut toujours redéfinir (polymorphisme de redéfinition ou d'héritage).

`public String toString()`

La chaîne retournée est constituée du nom de la classe de l'objet référencé par `this`, suivi de `@`, suivi de la clé identifiant l'instance

```
{  
    Animal a = new Animal(50) ;  
    System.out.println(a) ;  
}
```

Si la méthode n'est pas redéfinie dans la classe `Animal`, alors l'affichage suivant est produit :

Animal@13721b2

`public boolean equals(Object o)`

Compare l'objet référencé par `this` avec l'objet référencé par `o` :

- retourne `true` si `this` et `o` référencent le même objet (`this == o`).
- retourne `false` sinon

Méthodes de la classe Object()


Ces méthodes sont en général redéfinies car :

- `toString()` : on veut rendre plus expressif la chaîne de caractère associée à une instance d'une autre classe
- `equals(Object o)` : on ne cherche pas en général à tester l'égalité des références mais plutôt les données associées aux instances. Ainsi on veut que `equals()` renvoie `true` si les données de 2 instances comparées sont les mêmes (ex. 2 instances différentes qui présentent le même poids seront considérées comme identiques)

toString() - Redéfinition

```
public class Object
{
    ...
    ...
    public String toString()
    {
        ...
    }
}
```

```
public class Animal
{
    private double poids ;
    ...
    public String toString()
    {
        return("Poids="+this.poids);
    }
}
```



La méthode est bien redéfinie, puisque les méthodes ont la même signature et sont définies dans deux classes liées par une relation d'héritage.

Du coup, le code suivant

```
{
    Animal a = new Animal(50.) ;
    System.out.println(a) ;
}
```

Lors de l'évaluation, par liaison dynamique, c'est la méthode **toString()** de la classe **Animal** qui est appelée.

La méthode **toString()** héritée de la classe **Object** est masquée par sa redéfinition dans la classe **Animal**.

L'affichage produit sera :

Poids=50.

equals(Object o)-Redéfinition

Nous allons ici chercher à redéfinir une méthode `equals()` robuste pour la classe `Animal`.

Version 0

Sans redéfinition, c'est la méthode de la classe `Object()` qui est invoquée.

Le code de test suivant affiche `false` puis `true` puisque ce sont les références stockées dans les variables références qui sont comparées :

```
{
    Animal a1 = new Animal(50.) ;
    Animal a2 = new Animal( 50.) ;
    Animal a3 = a1 ;

    System.out.println(a1.equals(a2)) ;
    System.out.println(a1.equals(a3)) ;
}
```

Version 1

Avec la version suivante, le code test (*supra*) affiche 2 fois `true`.

```
public boolean equals(Animal a)
{
    return( this.poids == a.poids) ;
}
```

En effet, ce ne sont plus les références des variables références qui sont comparées, mais les valeurs des variables d'instance `poids` qui sont comparées. Les valeurs de cette variable sont identiques dans toutes les instances (il y en a 2) et vaut `50.`.

Le problème est qu'avec cette version 1, le code de test suivant affiche `false`. Ici la variable `a1` référence une instance d'`Animal` mais la variable est déclarée de type `Object`.

```
{
    Object a1 = new Animal(50.) ;
    Animal a2 = new Animal( 50.) ;

    System.out.println(a1.equals(a2)) ;
}
```

`a1` étant une variable référence sur un `Object`, c'est dans la classe `Object` que le compilateur recherche l'existence d'une méthode `equals(Animal)`. Il trouve la méthode `equals(Object)` compatible avec l'appel `a1.equals(a2)`. La liaison statique fixe le nom et le type des paramètres. A l'exécution la machine virtuelle recherche donc dans la classe la plus spécifique : `Animal` la méthode `equals(Object)` fixée à la compilation. Celle-ci n'y étant pas la méthode est recherchée dans la classe `Object`.

Ici, malheureusement la méthode `equals()` de la classe `Animal` n'est donc pas appelée et l'affichage est `false`

equals(Object o)-Redéfinition

Version 2 - Fin

```
public boolean equals(Object o)
{
    // Si o n'est pas une instance d'Animal ou une instance d'une
    // des classes filles d'Animal, o ne peut être équivalent à this

    if ( ! ( o instanceof Animal ) )
        return false ;

    // comme o est une instance d'Animal ou d'une de ses classes filles
    // on peut caster l'instance en Animal.

    Animal a = (Animal) o ;

    // Ainsi a et this sont des instances d'Animal et ont les mêmes
    // caractéristiques (méthodes et variables)
    // Ces 2 instances ont une méthode getPoids() reconnue par le
    // compilateur (ce que n'avait pas l'instance o (une instance
    // de Object n'a pas de méthode getPoids()))

    return( a.getPoids() == this.getPoids() ) ;
}
```

Remarque

La valeur de retour n'est ici calculée que sur la comparaison des valeurs d'une seule variable d'instance, la variable **poids**. Si la classe avait plusieurs variables d'instances, il aurait fallu comparer les valeurs de chacune de ces variables une à une dans une expression booléenne (tests liés par un ET logique).

Le modèle générique pour **MaClasse** et donc le suivant :

```
public boolean equals(Object o)
{
    if ( ! ( o instanceof MaClasse ) )
        return false ;
    MaClasse m = (MaClasse) o ;

    return(    m.getVar1() == this.getVar1() &&
              m.getVar2() == this.getVar2() &&
              ...
              m.getVarN() == this.getVarN() ) ;
}
```


Composition des equals()

Lorsque la classe en développement est la classe fille d'une autre classe, il faut impérativement se contraindre à utiliser un appel à la méthode `equals()` de la classe mère dans la définition de la méthode `equals()` de la classe fille.

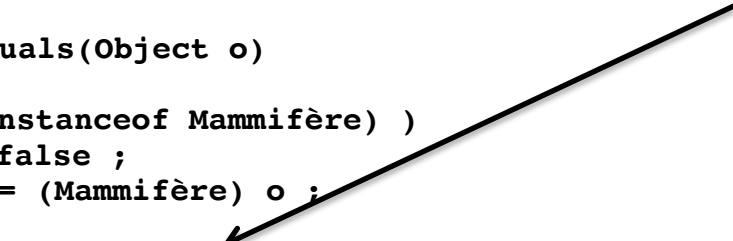
Exemple :

```
public class Mammifère extends Animal
{
    private int nbMammelles ;

    ...

    public boolean equals(Object o)
    {
        if ( ! ( o instanceof Mammifère) )
            return false ;
        Mammifère m = (Mammifère) o ;

        return(  super.equals(m)  &&
                this.getNbMammelles() == m.getNbMammelles() ) ;
    }
}
```



L'idée est

- de déléguer à la méthode `equals()` de la classe `Animal` la comparaison de l'ensemble des données du `Mammifère` héritées de la classe `Animal`.
- De ne rajouter dans la méthode `equals()` de la classe `Mammifère` que ce qui est spécifique à cette classe.

Cast

"caster" une expression revient à signaler au compilateur qu'elle est d'un type différent par rapport au type déclaré

```
{
    // Pour le compilateur o référence une instance de
    // la classe Object

    Object o = new Animal() ;

    // L'instruction suivante est rejetée par le compilateur car
    // o qui est déclaré comme Object est plus général qu'un Animal

    Animal a = o ;

    // Pour avoir le droit de faire cela il faut caster (redéfinir
    // le type déclaré)

    Animal a = (Animal) o ;
}
```

Exemple d'utilisation :

```
{
    Object o = new Animal() ;

    // L'instruction suivante est rejetée par le compilateur
    // lors de la liaison statique car o qui est déclaré comme
    // Object n'a pas de méthode getPoids()

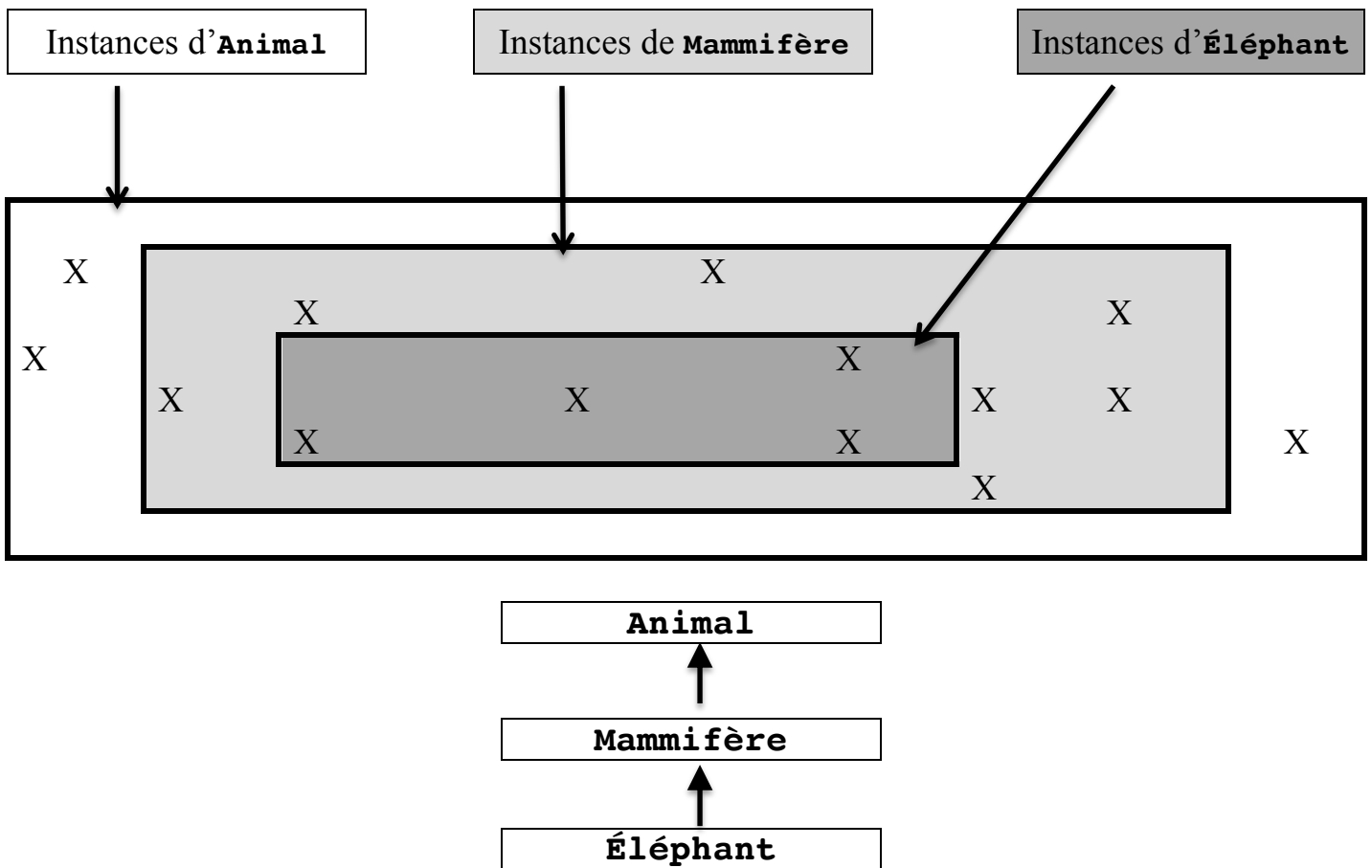
    double p1 = o.getPoids() ;

    // Pour avoir le droit de faire cela il faut caster (redéfinir
    // le type déclaré)

    double p2 = ((Animal) o).getPoids() ;
}
```

Le cast **(Animal)** signale au compilateur que dans la dernière instruction la référence **o** sur un **Object** doit être considérée comme une référence sur un **Animal**. Attention les parenthèses englobant **o** sont nécessaires.

instanceof



Soit le code suivant :

```
{  
    Mammifère m = new Mammifère( 500., 6);  
}
```

`m instanceof Animal` retourne `true` si la référence `m` désigne une instance d'`Animal`, et `false` sinon.

`m instanceof Object` retourne `true`

`m instanceof Animal` retourne `true`

`m instanceof Mammifère` retourne `true`

Exercices #3

VARIABLES DE CLASSE ET MÉTHODES DE CLASSE ET CONSTANTES

Ces exercices s'appuient sur les classes **Vaisseau** et **VolEnFormation** développées à l'issue du TP2. Vous pouvez télécharger l'archive `seance2_fin.tgz` contenant ces classes à l'adresse suivante :

<https://www.lipn.univ-paris13.fr/~santini/RN3/>

Chaque vaisseau est doté d'un numéro de châssis. Celui-ci est un identifiant unique qui ne peut être modifié. Nous nous proposons d'implémenter cet élément.

Question 1 :

Définissez une constante d'instance **NUMÉRO_DE_CHÂSSIS** qui définira le numéro du châssis et ses accesseurs si elle doit en avoir

Pour définir le numéro de châssis de chaque instance nous allons introduire une variable de classe **prochainNuméroDeChâssis** de **Vaisseau** et ses accesseurs.

Lors de chaque instantiation, cette variable de classe sera :

- 1/ lue. La valeur lue sera attribuée comme numéro de châssis à la nouvelle instance
- 2/ incrémentée de façon à ce qu'à l'instanciation suivante, le numéro de châssis soit différent et augmenté de +1.

Question 2 :

Définissez la variable de classe **prochainNuméroDeChâssis** et ses accesseurs.

Question 3 :

Définissez la méthode d'accès à la variable de classe **getProchainNuméroDeChâssis()**.

Question 4 :

Définissez la méthode d'accès à la variable de classe

incrémenteProchainNuméroDeChâssis() qui permet d'incrémenter la variable de classe.

Question 5 :

Modifiez le code des constructeurs pour que l'attribution des numéros de châssis soit prise en compte lors de l'instanciation.

Question 6 :

Modifiez la méthode **toString()** pour qu'elle intègre le numéro de châssis.

Question 7 :

Si ce n'est pas déjà fait définissez un programme de test pour vérifier le bon fonctionnement de ces nouvelles fonctionnalités.

Héritage

On souhaite définir une classe **CoordonnéesSpatiale** décrivant un point dans un espace à 3 dimensions qui utiliserait la classe **PointPlan** que l'on a l'habitude de manipuler: Pour définir la classe **CoordonnéesSpatiale** on s'appuie sur le principe d'héritage. Un premier essai conduit à l'implémentation d'une classe **CoordonnéesSpatiale** suivante.

```
public class CoordonnéesSpatiale extends PointPlan
{
    private double cote ;

    public Point3D() {
        super() ;
    }

    public Point3D(double x, double y, double z) {
        this.abscisse = x ;
        this.ordonnée = y ;
        this.cote      = z ;
    }

    public Point3D(Point3D p) {
        this.abscisse = p.abscisse ;
        this.ordonnée = p.ordonnée ;
        this.cote      = p.cote ;
    }

    public double getAbscisse() {
        return this.abscisse ;
    }

    public double getOrdonnée() {
        return this.ordonnée ;
    }

    public double getCote() {
        return this.cote ;
    }
}
```

Question 8 : Énumérez toutes les anomalies de conception de la classe **CoordonnéesSpatiale** ainsi que les instructions rejetées par le compilateur. Justifier chaque réponse.

Pour la suite du TP vous devez récupérer les fichiers définissant les classes **Passager**, **Mécanicien** et **Pilote**. Ces fichiers sont disponibles en téléchargement dans l'archive `seance3_debut.tgz` sur le site :

<https://www.lipn.univ-paris13.fr/~santini/RN3/>

Question 9 :

Écrivez une classe **TestPersonne** pour :

- Créer deux passagers, changer le numéro de billet de l'un d'eux et changer l'âge de l'autre
- Créer deux mécaniciens, transférer l'un d'eux dans un nouveau service et changer l'âge de l'autre
- Créer deux pilotes, ajouter 2 heures de vol à l'un d'eux et changer l'âge de l'autre.

Question 10 :

Identifiez ce que les classes **Mécanicien** et **Pilote** ont en commun.

Question 11 :

Une classe **Personnel** permettra de regrouper (ou factoriser) des propriétés de ces deux classes. Est-ce que ça a un sens de le faire ?

Question 12 :

Écrire une classe **Personnel** pour généraliser **Mécanicien** et **Pilote**.

Question 13 :

Proposez le code modifié des classes **Mécanicien** et **Pilote** pour qu'elles tirent partie de l'héritage de la classe **Personnel**.

Question 14 :

Proposez une hiérarchie de classe pour prendre en compte la classe **Passager**.

Question 15 :

Définissez la classe **Personne** et modifiez le code des classes **Passager**, et **Personnel** pour tenir compte de cette nouvelle hiérarchie.

Polymorphisme

On rappelle ici que l'ensemble des classes hérite de la classe **Object** certaines méthodes génériques. Notamment, parmi celles-ci, elles héritent de la méthode `equals(Object)` qui renvoie `true` si les deux instances ont la même référence et `false` sinon.

Question 16 :

Quels sont les éléments qui définissent la signature d'une méthode.

Considérons la classe **Personne** suivante :

```
public class Personne {
    private String nom ;
    private int age ;

    public Personne(String unNom, int unAge) {
        this.nom = unNom;
        this.age = unAge ;
    }
}
```

Et sa classe de test :

```
public class TestPersonne {
    public static void main(String[] args) {
        Personne p1 = new Personne("Toto", 30) ;
        Personne p2 = new Personne("Toto", 30) ;
        if(p1.equals(p2))
            System.out.println("EGALES") ;
        else
            System.out.println("DIFFERENTES") ;
    }
}
```

Question 17 :

Qu'affiche le programme de test ? Pourquoi ? Dessinez les références et les instances.

Question 18 :

Qu'affiche le programme de test si la méthode suivante est rajoutée dans la classe **Personne**. Justifiez votre réponse.

```
public boolean equals(Personne p)
{
    return (this.nom.equals(p.nom) && this.age==p.age) ;
}
```

Question 19 :

Qu'affiche le programme de test si ce dernier est modifié de la façon suivante.

```
public static void main(String[] args) {
    Personne p1 = new Personne("Toto", 30) ;
    Object p2 = new Personne("Toto", 30) ;
    if(p1.equals(p2))
        System.out.println("EGALES") ;
    else
        System.out.println("DIFFERENTES") ;
}
```

Question 20 :

Qu'affiche le programme de test si ce dernier est modifié de la façon suivante.

```

public static void main(String[] args)
{
    Personne p1 = new Personne("Toto", 30) ;
    Object p2 = new Personne("Toto", 30) ;
    if(p1.equals((Personne)p2))
        System.out.println("EGALES") ;
    else
        System.out.println("DIFFERENTES") ;
}

```

Question 21 :

Qu'affiche le programme de test si ce dernier est modifié de la façon suivante.

```

public static void main(String[] args)
{
    Object p1 = new Personne("Toto", 30) ;
    Personne p2 = new Personne("Toto", 30) ;
    if(p1.equals(p2))
        System.out.println("EGALES") ;
    else
        System.out.println("DIFFERENTES") ;
}

```

Question 23 :

Proposez une implémentation correcte de la méthode `equals(...)` pour la classe `Personne`.

Exercices optionnels

Soient les 3 classes suivantes (données dans l'archive de départ) :

```

public class Vaisseau
{
    public void départ()
    {
        System.out.println("Je parts.");
    }
}

```

```

public class VaisseauAVoile extends Vaisseau
{
    public void départ()
    {
        System.out.println("Je mets les voiles.");
    }

    public void percute() {
        System.out.println("Je coule.");
    }
}

```



```

public class VaisseauDeLEspace extends Vaisseau
{
    public void départ()
    {
        System.out.println("Je lance les moteurs a fusion.");
    }

    public void percute()
    {
        System.out.println("Je dépressurise.");
    }
}

```

Soit le programme principal suivant :

```

1  public class TestVaisseau2 {
2      public static void main(String[] args) {
3
4          Vaisseau v1 ;
5          v1 = new Vaisseau() ;
6          v1.départ() ;
7          v1.percute() ;
8          v1 = new VaisseauDeLEspace() ;
9          v1.départ() ;
10         v1.percute() ;
11
12         VaisseauDeLEspace v2 = new VaisseauDeLEspace() ;
13         v2.départ() ;
14         v2.percute() ;
15         v2 = new Vaisseau() ;
16         v2.départ() ;
17         v2.percute() ;
18
19         v1 = (Vaisseau)( v2 ) ;
20         v1.départ() ;
21         v1.percute() ;
22
23         v2 = (VaisseauDeLEspace)( v1 ) ;
24         v2.départ() ;
25         v2.percute() ;
26
27         Vaisseau [] flotte=new Vaisseau [3];
28         flotte[0] = new VaisseauDeLEspace();
29         flotte[1] = new Vaisseau();
30         flotte[2] = new VaisseauAVoile();
31
32         for (int i = 0 ; i < flotte.length; i++)
33             flotte[i].départ();
34
35     }
36 }

```

Question 24 :

Dans le programme de test ci-dessus, quelles instructions posent problème ?

Question 25 :

Sans les instructions problématiques, qu'afficherait le programme ?

Question 26 :

Dessinez l'arbre d'héritage complet des 3 classes.

Question 27 :

Redéfinissez les méthodes de la classe **Object** pour chacune des 3 classes de Vaisseau.

Question 28 :

Modifiez la classe **VolEnFormation** pour qu'elle admette un nombre de Vaisseau compris entre 2 et 20 inclus.

Question 29 :

Il est possible de définir un vol en formation avec des **VaisseauAVoile**. Vous le testerez en proposant un programme de test définissant :

- une formation de 4 **Vaisseau**
- avec 2 **Vaisseau** génériques
- un **VaisseauDeLEspace**
- un **VaisseauAVoile**

Dans ce programme vous appliquerez les méthodes **départ()** et **percute()** à tous les vaisseaux de la formation. Est-ce possible ?

Question 29 :

Vérifiez que le constructeur par copie de la classe **VolEnFormation** fonctionne correctement.