

# Programmation Orientée Objet

---

La programmation Orientée Objet est une des concepts principaux du langage Java.

La réalisation de classes est la principale possibilité offerte par le langage pour définir de nouveaux types.

## Qu'est ce qu'une classe ?

Une classe peut être assimilée à un plan logiciel ou à un prototype pour les objets. Elle définit les éléments qui donneront une existence à un objet.

On y trouvera :

- des propriétés
- des méthodes pour manipuler ces propriétés.

Les propriétés seront un ensemble de variables typés qui posséderont des valeurs variant tout le long du cycle de vie de l'objet. **Les valeurs des propriétés représentent l'état de l'objet**

Les méthodes permettent d'accéder aux propriétés, de modifier leur valeur ou de réaliser un comportement en manipulant l'ensemble des valeurs des propriétés.

La classe en tant que telle n'a pas d'existence réelle. Pour donner vie à une classe et l'attribuer à des variables, il faut créer une instance de celle-ci, c'est à dire créer un **objet**.

Plane
+String brand +String model +int capacity +double speed +double elevation
+takeoff() +landing() +accelerate(double speed) +deccelerate(double speed) +addPassagers(int number)

## Définir une classe

Pour définir une classe,

- une classe est dans un fichier portant le même nom

- son nom commence par une majuscule et respecte le format pascalCase.
- elle est définie par le mot clé **class**
- elle possède des propriétés, des constructeurs et des méthodes.

```
public class AirPlane {  
  
    // 1 déclaration des attributs / propriété  
    private String brand;  
    private String model;  
    private int capacity;  
    private double speed;  
    private double elevation;  
  
    //Les méthodes  
    public void decelerate(double speedValue) {  
        if (this.speed - speedValue > 0) {  
            this.speed -= speedValue;  
        }  
    }  
  
    public void accelerate(double speedValue) {  
        this.speed += speedValue;  
    }  
}
```

## Les variables

Élément « nommé » permettant de stocker des informations sur la classe ou sur l'instance de la classe (objet)

Le tableau suivant présente les types de variables que vous pouvez rencontrer dans une classe.

Variables locales	Variables d'instance	Variables de classe
<ul style="list-style-type: none"><li>• Variables déclarées à l'intérieur des méthodes de la classe ou entre accolades {...}</li><li>• Ces variables ne sont accessibles uniquement à l'intérieur des méthodes</li></ul>	<ul style="list-style-type: none"><li>• Variables déclarées à l'intérieur d'une classe mais pas à l'intérieur d'une méthode de la classe.</li><li>• La valeur des variables représente l'état de la classe à un instant t.</li><li>• La valeur de ces variables est spécifique à l'instance de la classe</li></ul>	<ul style="list-style-type: none"><li>• Variables déclarées à l'intérieur d'une classe avec le mot clé <b>static</b></li><li>• La valeur de ces variables sont partagées entre toutes les instances de la classe</li><li>• On utilise ces variables pour représenter des constantes.</li><li>• En fonction de la portée, il est possible d'accéder à ces variables avec la classe</li></ul>

Concernant les variables d'instances, chaque instance possédera « ses » propres valeurs pour chaque propriété

Pour créer une instance, il faut utiliser le mot clé **new**

## Les constructeurs

Un constructeur est un bloc d'instructions permettant d'initialiser une nouvelle instance d'une classe.

Un constructeur :

- porte le **même nom que la classe**
- retourne aucun type.
- accède au mot clé **this** qui représente l'instance de la classe
- sert à configurer l'état initial de l'instance

Une classe peut ne pas avoir de constructeur. Dans ce cas, Java configure automatiquement un constructeur par défaut ne réalisant aucun traitement. Les propriétés de l'instance seront initialisées par les valeurs par défaut.

Dès qu'un constructeur est défini, le constructeur par défaut n'est plus accessible.

On peut définir autant de constructeurs que l'on souhaite. La différence se fera sur le nombre et le type des paramètres les définissant.

Pour notre classe **AirPlane**, le constructeur suivant initialisera la marque et le modèle. Quant aux autres propriétés (speed...), elles seront initialisées par les valeurs par défaut.

```
public AirPlane(String brand, String model) {  
    this.brand = brand;  
    this.model = model;  
}
```

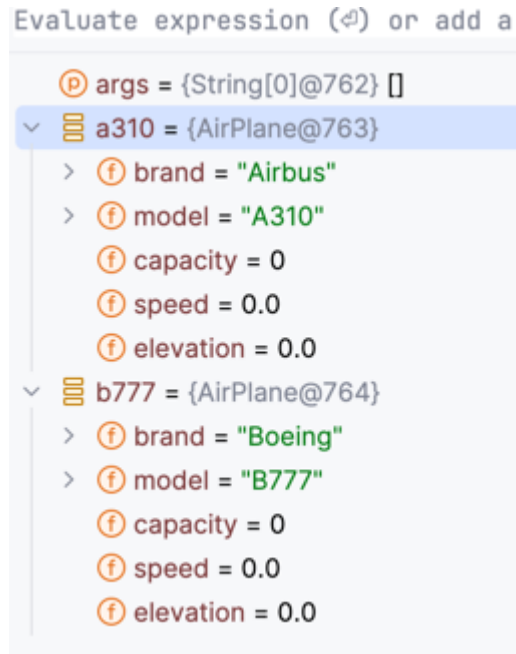
le mot clé **this** permet d'accéder à l'instance de la classe c'est à dire à l'objet que nous sommes entrain d'initialiser. Suivi du `.`, il permet d'accéder aux propriétés et aux méthodes de l'instance. le mot clé **this** sera disponible dès que l'on accède à des méthodes d'instance.

La création d'une instance se fera par l'utilisation du mot clé **new** suivi du constructeur souhaité.

L'exemple suivant permet de définir 2 variables : a310 et b777. Ces 2 variables sont du type **AirPlane**. Puis, nous initialisons les variables en appelant le constructeur prenant la marque et le modèle.

```
public class Main1 {  
  
    public static void main(String[] args) {  
        AirPlane a310 = new AirPlane("Airbus", "A310");  
        AirPlane b777 = new AirPlane("Boeing", "B777");  
    }  
}
```

Le mode debug de l'IDE permet de visualiser ces 2 instances avec leurs propriétés respectives.



Le tableau suivant synthétise les 3 types de constructeurs :

Constructeur par défaut	Constructeur sans paramètre	Constructeur avec paramètres
<ul style="list-style-type: none"> <li>En l'absence de constructeur, la classe possède, par défaut, un constructeur initialisant les variables d'instance à leur valeur par défaut</li> <li>Dans le cas où un constructeur est défini dans la classe, le constructeur par défaut ne sera plus disponible</li> </ul>	<ul style="list-style-type: none"> <li>Constructeur définie dans la classe et ne prenant aucun paramètre</li> </ul> <pre> public AirPlane() {     System.out.println("Initialisation")     this.model = "inconnu"; }           </pre>	<ul style="list-style-type: none"> <li>Constructeur dans la classe permettant et prenant des paramètres</li> </ul> <pre> public AirPlane(String brand, String model) {     this.brand = brand;     this.model = model; }           </pre>

Il est tout à fait possible de déclarer autant de constructeurs que l'on souhaite. Il s'agit du principe de l'overloading.

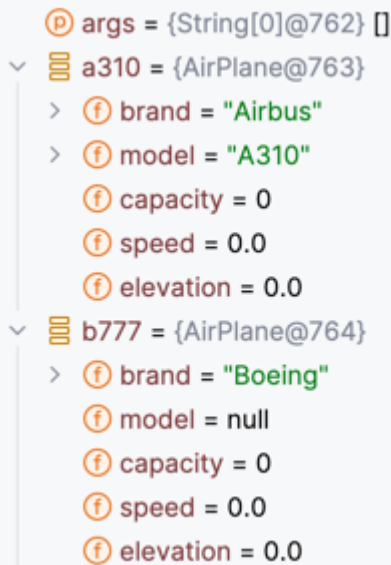
En ajoutant un constructeur avec uniquement la marque, nous pouvons créer des instances à partir de ce dernier.

```

public AirPlane(String brand) {
    this.brand = brand;
}
  
```

```
public class Main2 {  
  
    public static void main(String[] args) {  
        AirPlane a310 = new AirPlane("Airbus", "A310");  
        AirPlane b777 = new AirPlane("Boeing");  
    }  
}
```

Le mode debug permet de visualiser que la variable *b777* ne possède que la marque.



```
args = {String[0]@762} []  
a310 = {AirPlane@763}  
  > brand = "Airbus"  
  > model = "A310"  
    capacity = 0  
    speed = 0.0  
    elevation = 0.0  
b777 = {AirPlane@764}  
  > brand = "Boeing"  
    model = null  
    capacity = 0  
    speed = 0.0  
    elevation = 0.0
```

## Les méthodes

Les méthodes d'instance vont permettre d'accéder à notre objet afin qu'il réalise un traitement. Ce traitement pourra modifier l'état de notre objet via la modification de la valeur des propriétés d'instance.

L'accès à ces méthodes se fera en utilisant le point.

L'exemple présente 2 méthodes qui vont agir sur la vitesse de l'instance.

```
public void decelerate(double speedValue) {  
    if (this.speed - speedValue > 0) {  
        this.speed -= speedValue;  
    }  
}  
  
public void accelerate(double speedValue) {  
    this.speed += speedValue;  
}
```

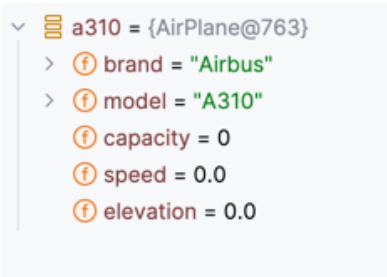

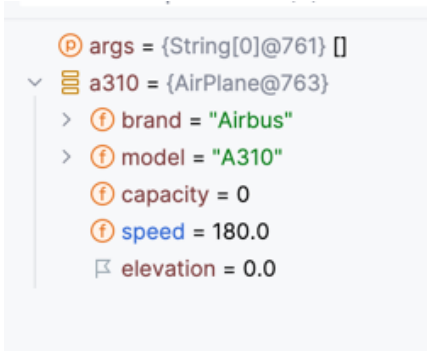
```
public class Main3 {
```

```

    public static void main(String[] args) {
        AirPlane a310 = new AirPlane("Airbus", "A310");
        a310.accelerate(200);
        a310.decelerate(20);
    }
}

```

Le tableau montre l'évolution de la vitesse de l'instance **a310**.

Etat 1	Etat 2	Etat 3
 <pre> a310 = {AirPlane@763}   &gt; brand = "Airbus"   &gt; model = "A310"   capacity = 0   speed = 0.0   elevation = 0.0 </pre>	 <pre> args = {String[0]@761} [] a310 = {AirPlane@763}   &gt; brand = "Airbus"   &gt; model = "A310"   capacity = 0   speed = 200.0   elevation = 0.0 </pre>	 <pre> args = {String[0]@761} [] a310 = {AirPlane@763}   &gt; brand = "Airbus"   &gt; model = "A310"   capacity = 0   speed = 180.0   elevation = 0.0 </pre>

## Les accesseurs

La POO repose sur le principe d'encapsulation. Ce principe permet de masquer les détails d'un objet à un client. Afin d'accéder aux propriétés, des méthodes d'accès sont créées afin de retourner ou de modifier les propriétés de l'instance

Ainsi,

- Les propriétés sont **privée** le plus souvent. Lors de leur déclaration, les propriétés de la classe sont préfixés par **private**.
- Les propriétés privées ne sont accessible que les méthodes de l'instance
- L'accès et la modification des valeurs des propriétés seront le plus souvent réalisé par des méthodes appelées **getter** et **setter**.

En complétant la classe **AirPlane**, l'ajout d'un getter et d'un setter sur la propriété **model** permet de lui associer une valeur et d'y accéder.

```

public void setModel(String model) {
    this.model = model;
}

public int getCapacity() {
    return capacity;
}

```

```
public class Main4 {  
  
    public static void main(String[] args) {  
        AirPlane b777 = new AirPlane("Boeing");  
        b777.setModel("B777");  
        System.out.println("L'instance b777 est associé au modele : " +  
b777.getModel());  
    }  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Main4  
L'instance b777 est associé au modele : B777
```

Une erreur serait de mettre des setter sur toutes les classes. Hors, certaines propriétés ne doivent pas être modifiées sans contrôle. Par exemple, la propriété **speed** ne doit être modifiée directement vu qu'elle est contrôlée par les méthodes **accelerate** et **decelerate**.

On préfère donc ne permettre simplement la récupération de la valeur de la propriété **speed**.

```
public double getSpeed() {  
    return speed;  
}
```

```
public class Main5 {  
  
    public static void main(String[] args) {  
        AirPlane a310 = new AirPlane("Airbus", "A310");  
        a310.accelerate(200);  
        a310.decelerate(20);  
        System.out.printf("Avion %s %s a une vitesse de %.2f \n",  
a310.getBrand(), a310.getModel(), a310.getSpeed());  
    }  
}
```

```
mvn --quiet compile exec:java -Dexec.mainClass=Main5  
L'instance b777 est associé au modele : B777
```

## L'héritage

L'héritage est la capacité d'une classe d'être créée à partir d'une autre classe en l'étendant. Ce principe permet d'éviter de dupliquer le code et de factoriser le code en commun dans une classe dite mère.

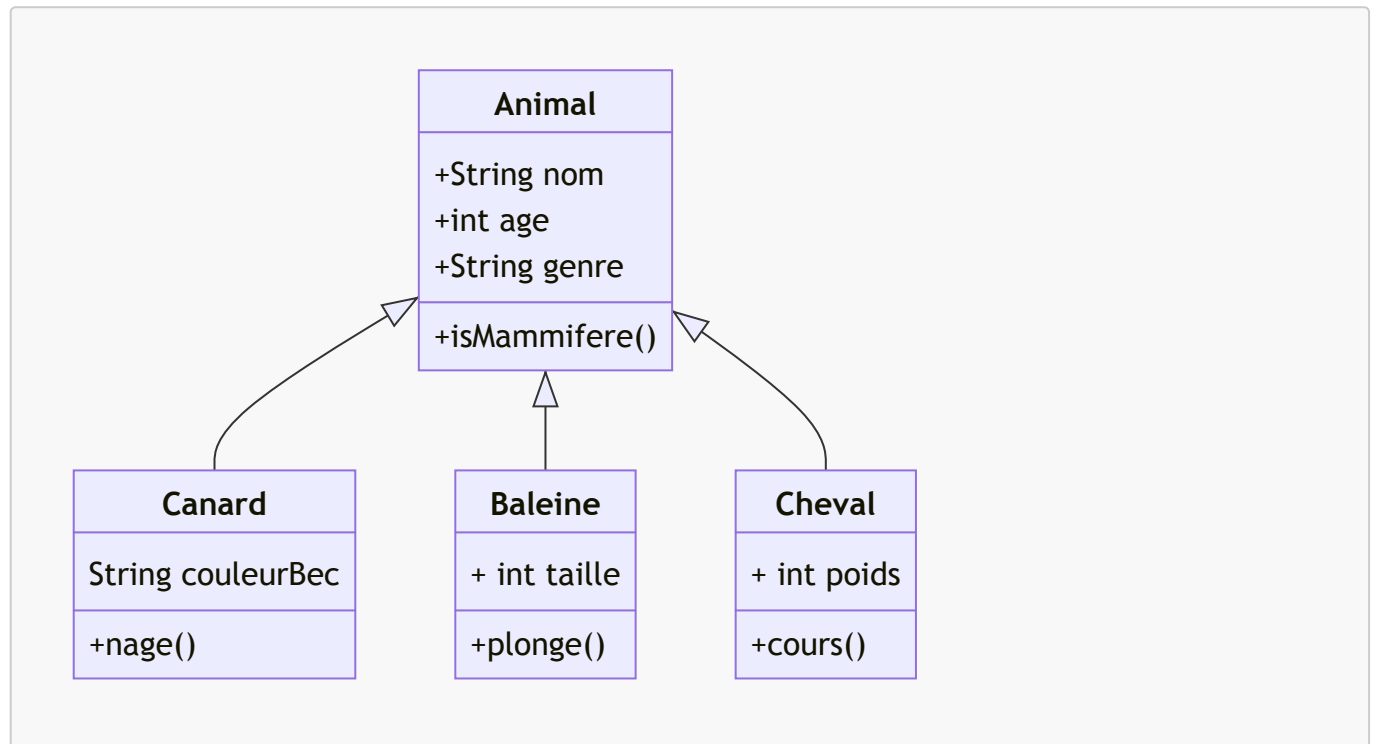
Les classes héritantes d'une autre classe héritent des comportements de la classe héritée.

Les classes héritante sont dites des classes filles et peuvent définir :

- de nouvelles propriétés
- de nouvelles méthodes.

Les classes fille peuvent également modifier le comportement de la classe mère en redéfinissant les méthodes de la classe mère (**overriding**)

Exemple d'héritage :



Comment hériter ?

En Java, une classe peut hériter d'une autre classe en utilisant le mot clé **extends**. Le mot clé **extends** se place après le nom de classe et est suivi du nom de classe mère.

Attention, une classe **ne peut hériter que d'une seule classe !**

Dans le cas des animaux :

```
public class Animal {
    private String nom;
    private int age;
    private String genre;

    public boolean isMammifere(){
        return false;
    }
}

public class Baleine extends Animal{
```



```
private int taille;

public void plonge(){

}

}
```

Ici, nous constatons que la classe **Baleine** représente un mammifère. La méthode **isMammifere** de la classe doit être redéfinie dans la classe **Baleine**.

Ainsi, la classe **Baleine** devient :

```
public class Baleine extends Animal{

    private int taille;

    public void plonge(){

    }

    @Override
    public boolean isMammifere() {
        return true;
    }

}
```

Ici, vous voyez apparaître l'annotation `@Override`. Les annotations sont énormément utilisées dans le cadre de développement d'applications d'entreprise. Elles permettent d'ajouter des comportements transverses sans que vous soyez obligés de coder. L'annotation `@Override` est purement indicative et n'a aucun apport lors de l'exécution.

## Le mot clé super

Au sein des classes fille, nous pouvons redéfinir les méthodes. Cette redéfinition peut cependant s'appuyer/utiliser ce qui a été initialement défini dans la classe mère.

Le mot clé **super** permet d'appeler les méthodes de la classe mère au sein d'une classe fille,

Par exemple, ajoutons une méthode **getNom** dans la classe **Animal**. Cette méthode permettra de fournir le nom de l'animal.

Dans le cas de la classe **Baleine**, nous souhaitons que la méthode **getNom** retourne le nom de la baleine en préfixant pas *Baleine*.

En recodant, nous obtenons :

```
public abstract class Animal {

    private String nom;
    private int age;
```

```
private String genre;

public Animal(String nom, int age) {
    this.nom = nom;
    this.age = age;
}

public String getNom() {
    return nom;
}

}

public class Baleine extends Mammifere{

    private int taille;

    public Baleine(String nom, int age) {
        super(nom, age);
    }

    @Override
    public String getNom() {
        return "Baleine " + super.getNom();
    }

    public void plonge(){
    }

}
```

```
import animal.Baleine;

public class Main10 {
    public static void main(String[] args) {
        var baleine = new Baleine("Moby Dick", 173);
        System.out.println(baleine.getNom());
    }
}
```

Baleine Moby Dick

## Le mot clé **abstract**

Dans l'exemple ci dessus, aucune obligation n'a été imposé quant à la redéfinition de la méthode **isMammifere**. La classe **Balein** aurait très bien pu ne pas redéfinir cette méthode et d'un point de vue compilation et exécution, aucune erreur n'aurait été rencontrée.

Et pourtant, conceptuellement, une **baleine** est un mammifère.

Le moté clé **abstract** permet de forcer la redefinition des méthodes dites abstraites par les classes héritantes. Il doit être positionné avec le mot clé **class** de la classe mère puis peut être utilisé lors de la déclaration de certaines méthodes.

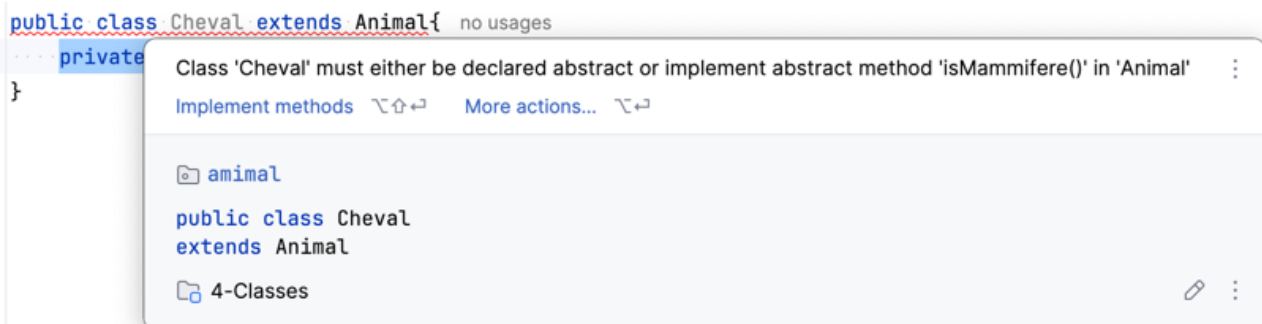
Ces méthodes abstraites ne fourniront aucun bloc de code dans la classe. Seule la signature sera définie (ce que la méthode retourne et ce qu'elle prend en paramètre).

Les classes **filles devront** implémenter les méthodes abstraites. C'est à dire fournir un bloc d'instructions.

Voici la nouvelle déclaration de la classe **Animal** :

```
public abstract class Animal {  
  
    private String nom;  
    private int age;  
    private String genre;  
  
    public abstract boolean isMammifere();  
}
```

Si nous ne faisons aucune autre modification, le projet ne compile plus.



Il est nécessaire de définir une implémentation de cette méthode dans les classes **Cheval** et **Canard**

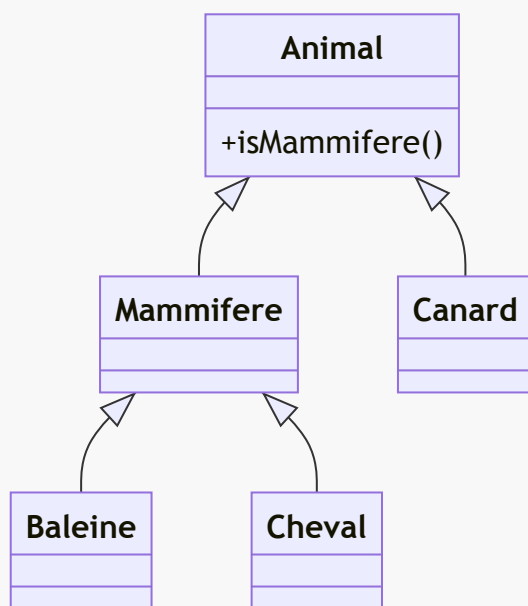
```
public class Cheval extends Animal{  
    private int poids;  
  
    @Override  
    public boolean isMammifere() {  
        return true;  
    }  
}  
  
public class Canard extends Animal {  
  
    private String couleurBec;  
  
    public void nage() {  
    }  
}
```

```
@Override
public boolean isMammifere() {
    return false;
}
```

Le mot clé final.

La méthode **isMammifere** va devoir être implémentée dans l'ensemble des classes représentant des animaux. Dans notre exemple, les classes **Cheval** et **Baleine** représentent des animaux tous 2 des mammifères et potentiellement partageant des caractéristiques communes.

Ainsi, une nouvelle classe **Mammifere** peut être réalisée afin de regrouper les mammifères entre eux.



Le nouveau diagramme de classe présente le nouvel arbre d'héritage.

Dés lors, la méthode **isMammifere** sera définie dans la classe **Mammifere** et ne devra plus être redéfinie par les classes héritant de la classe **Mammifere**.

Pour éviter qu'une méthode ne soit redéfinie, il suffit de la faire précéder par le mot clé **final**. Le mécanisme d'**overriding** ne pourra plus avoir lieu.

La classe **Mammifere** devient :

```
public class Mammifere extends Animal{
    @Override
    public final boolean isMammifere() {
        return false;
    }
}
```

Si nous changeons l'héritage sur la classe **Baleine** en laissant la méthode **isMammifere**.

```
public class Mammifere extends Animal{
    @Override
    public final boolean isMammifere() {
        return true;
    }
}
```

```
public class Baleine extends Mammifere{ no usages
```

```
... private int taille; no usages
```

```
... public void plonge(){ no usages
... }
```

```
... @Override no usages
```

```
... public boolean isMammifere(){
```

```
... return true
```

```
... }
```

'isMammifere()' cannot override 'isMammifere()' in 'animal.Mammifere'; overridden method is final

Make 'Mammifere.isMammifere()' not final More actions...

© animal.Baleine

public boolean isMammifere()

Overrides: isMammifere in class Mammifere

4-Classes

La classe **Baleine** doit devenir :

```
public class Baleine extends Mammifere{

    private int taille;

    public void plonge(){
    }

}
```

## La classe *java.lang.Object*

Toute classe étend directement ou indirectement de la classe *java.lang.Object* même si aucun héritage n'est indiqué.

Ainsi, les méthodes de la *java.lang.Object* sont disponibles.

Object
<div>+Object clone() +int hashCode() +boolean equals(Object obj) +String toString() +void finalize() +Class getClass() +void notify() +void notifyAll() +void wait() +void wait(long timeout) +void wait(long timeout, int nanos)</div>

Attardons nous sur les méthodes principales.

## La méthode toString

Cette méthode permet de proposer une représentation de l'objet sous la forme d'une chaîne de caractères.

L'implémentation de la classe **Object** est très sommaire. Elle ne fournit que le nom de la classe complète de la valeur retournée par la méthode **hashCode**.

```
import animal.Baleine;

public class Main6 {
    public static void main(String[] args) {
        var baleine= new Baleine("Moby Dick", 173);
        System.out.println(baleine);
    }
}
```

```
animal.Baleine@506e6d5e
```

la méthode `System.out.println` invoque directement la méthode **toString** lorsqu'elle reçoit un objet en paramètre.

En redéfinissant la méthode **toString**, il serait plus intéressant d'afficher le nom et l'âge de la baleine.

Pour cela, redéfinissons la méthode `toString` dans la classe **Animale**.

```
package animal;
```

```
import java.util.StringJoiner;

public abstract class Animal {

    private String nom;
    private int age;
    private String genre;

    public Animal(String nom, int age) {
        this.nom = nom;
        this.age = age;
    }

    public abstract boolean isMammifere();

    @Override
    public String toString() {
        return new StringJoiner(", ", this.getClass().getSimpleName() + "
[", "]")
            .add("age=" + age)
            .add("nom='" + nom + "'")
            .toString();
    }
}
```

En executant le programme précédent, l'affichage est désormais le suivant :

```
Baleine[age=173, nom='Moby Dick']
```

## La méthode equals

Cette méthode est fréquemment utilisée pour vérifier que 2 objets sont égaux.

Par défaut, la méthode fournie pour la classe **java.lang.Object** vérifie uniquement que les 2 objets sont la même instance.

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

Si nous laissons l'implémentation, par défaut, le code suivant indique que les 2 baleines sont différents. Hors, elles semblent cependant être identiques.

:

```
public class Main7 {  
    public static void main(String[] args) {  
        var baleine1 = new Baleine("Moby Dick", 173);  
        var baleine2 = new Baleine("Moby Dick", 173);  
        System.out.printf("Les baleines sont elle les mêmes ? Reponse :  
%b", baleine1.equals(baleine2));  
    }  
}
```

Résultat :

```
Les baleines sont elle les mêmes ? Reponse : false
```

L'égalité entre 2 objets dans la réalité n'est pas uniquement basée sur le fait que ce soit la même instance. Il s'agit le plus souvent d'une comparaison entre les valeurs des propriétés caractérisant chaque instance.

Par exemple, dans la classe **Animal**, on pourrait considérer que 2 objets sont égaux :

- soit les objets sont la même instance
- soit ils sont issus de la même classe et ont le même nom et le même age.

Ainsi, la classe **Animal** doit redéfinir la méthode **equals**.

```
public abstract class Animal {  
  
    private String nom;  
    private int age;  
    private String genre;  
  
    public Animal(String nom, int age) {  
        this.nom = nom;  
        this.age = age;  
    }  
  
    public abstract boolean isMammifere();  
  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (o == null || getClass() != o.getClass()) return false;  
        Animal animal = (Animal) o;  
        return age == animal.age && Objects.equals(nom, animal.nom);  
    }  
  
    //Autres méthode  
}
```



En exécutant la méthode **main** précédente, on constate que l'égalité devrait vraie.

```
Les baleines sont elle les mêmes ? Reponse : true
```

## La méthode hashCode

Cette méthode est utilisée par des algorithmes de hachage avant de vérifier que 2 objets sont égaux. Par exemple, la classe **HashSet** fréquemment utilisée lors de l'utilisation de **Set** utilise cette méthode avant de comparer les objets. Ainsi, des objets identiques doivent avoir en premier lieu le même résultat au niveau du Hashcode.

un set est une ensemble d'objets ne contenant pas de doublon.

Si nous conservons l'implémentation de la méthode **equals** mais que nous ne redéfinissons pas la méthode **hashCode**, nous constatons que le set contient **2 éléments**. Or, les 2 baleines étant les mêmes, un seul élément ne devrait être présent dans la set.

```
public class Main8 {
    public static void main(String[] args) {
        var baleine1 = new Baleine("Moby Dick", 173);
        var baleine2 = new Baleine("Moby Dick", 173);

        Set<Animal> set = new HashSet<>();
        set.add(baleine1);
        set.add(baleine2);

        System.out.printf("Nombre d'éléments : %d", set.size());
    }
}
```

```
Nombre d'éléments : 2
```

Il est donc nécessaire de redéfinir la méthode **hashCode**. L'implémentation de ce type méthode repose sur la réalisation d'une empreinte numérique en fonction des propriétés utilisées dans l'égalité. Pour la classe **Animal**, l'implémentation serait la suivante :

```
public abstract class Animal {

    private String nom;
    private int age;
    private String genre;

    public Animal(String nom, int age) {
        this.nom = nom;
        this.age = age;
    }
}
```

```
}

public abstract boolean isMammifere();

// methode equals

@Override
public int hashCode() {
    return Objects.hash(nom, genre);
}

// methode toString

}
```

Si nous relançons le main précédent, nous obtenons bien le résultat attendu :

Nombre d'éléments : 1

## L'agrégation / La composition / L'association

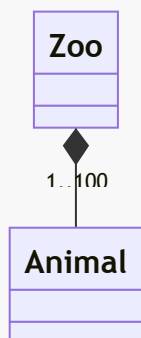
Comme décrit une classe permet de définir de nouveaux type, ces nouveaux types peuvent ainsi être utilisés pour déclarer des propriétés.

L'association est le fait qu'un objet soit lié à un objet d'une autre classe.

L'agrégation et la composition sont le fait qu'un objet soit lié à plusieurs objets d'une autre classe. La composition implique en plus une appartenance. La destruction de l'objet contenant détruit les objets liés.

Créons la classe **Zoo** qui va contenir un ensemble d'animaux. Cet ensemble est limité au maximum à 100 animaux.

Au niveau UML,



Au niveau de la classe **Zoo** :

```

public class Zoo {
    private Animal[] animaux;

    public Zoo() {
        this.animaux = new Animal[100];
    }

    public void addAnimal(Animal animal, int index){
        this.animaux[index] = animal;
    }

    @Override
    public String toString() {
        return new StringJoiner(", ", Zoo.class.getSimpleName() + "[",
" ]")
            .add("animaux=" + Arrays.stream(animaux)
                .filter(Objects::nonNull)
                .map(Object::toString)
                .collect(Collectors.joining(", ")))
            .toString();
    }
}

```

La méthode **addAnimal** permet d'ajouter des animaux au sein de la classe **Zoo**.

```

import animal.Baleine;
import animal.Canard;
import animal.Zoo;

public class Main9 {
    public static void main(String[] args) {
        Zoo zoo = new Zoo();
        zoo.addAnimal(new Baleine("Moby Dick", 173), 0);
        zoo.addAnimal(new Canard("Donald Duck", 90, "jaune"), 1);
        System.out.println(zoo);
    }
}

```

```

Zoo[animaux=Baleine[age=173, nom='Moby Dick'],Canard[age=90, nom='Donald
Duck']]

```

Vous pouvez constater que la méthode **addAnimal** prend 2 paramètres dont le premier est un paramètre de type **Animal**. Nous appelons cette méthode en passant une instance de la classe **Baleine** puis une instance de la classe **Canard**. Aucune erreur est levée ce qui est tout à fait normal, les classes **Baleine** et **Canard** héritent toutes 2 de la classe **Animal**. Donc, toute instance de ces classes sont, par héritage, des instances de la classe **Animal**. On parle ici de polymorphisme.