

Objectif de l'atelier

L'objectif de cet atelier est de comprendre le mot réservé « abstract »

.
Vous allez apprendre à:

- Le rôle de abstract lorsqu'il est appliqué à une classe
- Le rôle de abstract lorsqu'il est appliqué à une méthode
- L'obligation d'implémentation des méthodes abstraites chez les classes filles

Introduction

Une classe abstraite est quasiment comme une classe normale. Oui, comme une classe que vous avez maintenant l'habitude de coder.

Ceci dit, elle a tout de même une particularité : **vous ne pouvez pas l'instancier !**

Créer alors une classe A déclarée abstraite. Puis essayer de l'instancier dans une classe de Test contenant la méthode main.

Code Java de la classe Test.

```
1  public class Test{
2
3      public static void main(String[] args){
4
5          A obj = new A();//Erreur de compilation ! !
6
7      }
8  }
```

Formation Orienté Objet et Java de base

Atelier 6 : Abstract

Encadré par M.BOULCHAHOU

À quoi servent les classes abstraites ?

Pour répondre à cette question, Imaginez que vous êtes en train de réaliser un projet qui gère différents types d'animaux.

Créer un projet Java et nommer le « GestionAnimaux »

Dans ce projet, vous aurez :

- *des loups*
- *des chiens*
- *des chats*
- *des lions*
- *des tigres.*

Je pense tout de même que vous n'allez pas faire toutes vos classes bêtement... il va de soi que tous ces animaux ont des choses en commun ! Et qui dit chose en commun... dit héritage. Que pouvons-nous définir de commun à tous ces animaux, sinon :

- *une couleur*
- *un poids*
- *qu'ils crient*
- *qu'ils se déplacent*
- *qu'ils mangent*
- *qu'ils boivent*

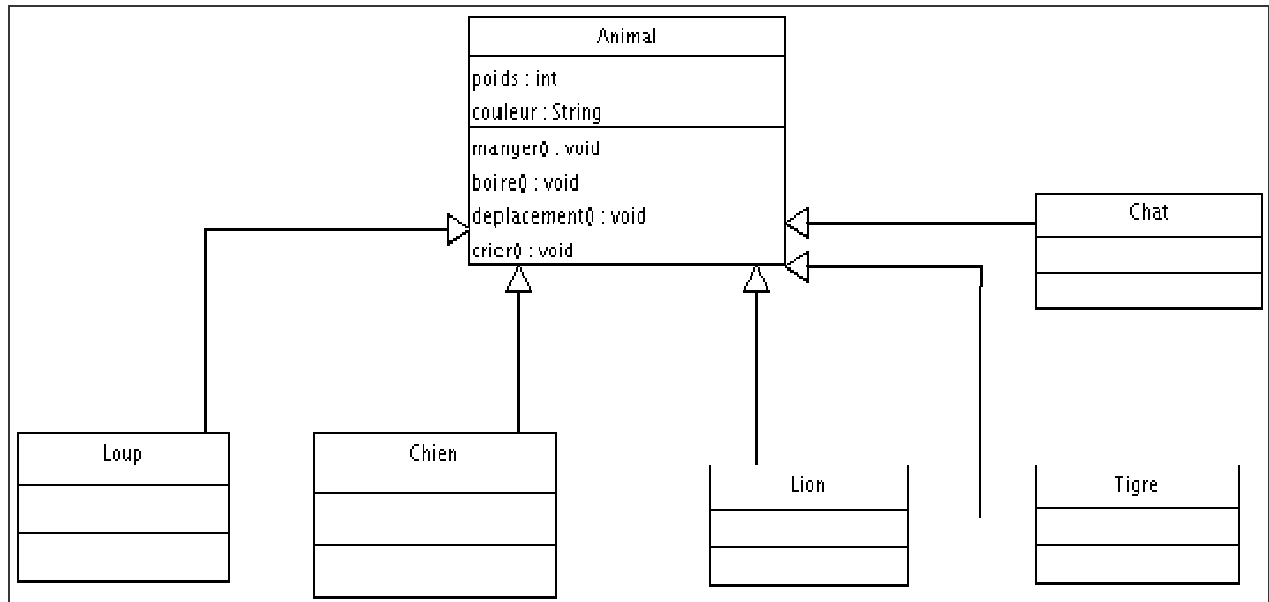
Nous pouvons donc créer une classe mère, appelons-la Animal.

Avec ce que nous avons dégagé de commun, nous pouvons lui définir des attributs et des méthodes. Voici donc à quoi pourraient ressembler nos classes pour le moment :

Formation Orienté Objet et Java de base

Atelier 6 : Abstract

Encadré par M.BOULCHAHOUB



Nous avons bien notre classe mère **Animal** et nos animaux qui en héritent. À présent, laissez-moi vous poser une question.

Vu que notre classe *Animal* est publique dans notre cas, qu'est-ce qu'est censé faire un objet *Animal* ? Quel est son poids, sa couleur, que mange-t-il ?

Créer une méthode `manger ()` dans la classe *Animal*. Et essayer de l'appeler dans la classe de *Test*.

Vous aurez un morceau de code qui ressemble à ceci :

```
1  public class Test{
2      public static void main(String[] args){
3          Animal ani = new Animal();
4          ani.manger();//Que doit-il faire ? ?
5      }
6  }
```

Je ne sais pas comment mange un objet *Animal*...

Vous conviendrez que toutes les classes ne sont pas bonnes à être instanciées !

Formation Orienté Objet et Java de base

Atelier 6 : Abstract

Encadré par M.BOULCHAOUB

C'est là que rentrent en jeu nos **classes abstraites**. En fait, ces classes servent à définir **un super classe** et obliger les classes filles à implémenter les méthodes abstraites de leur classe mère

Noter que même si les classes abstraites ne peuvent pas être instanciées, elles contiennent des constructeurs appelés à travers le `super ()` des classes filles

Une classe Animal abstraite

En fait, il existe une règle pour qu'une classe soit considérée comme abstraite. Elle doit être déclarée avec le mot clé **abstract**.

Voici un exemple illustrant mes dires :
Classe déclarée abstraite :

```
1  abstract class Animal{  
2  
3  }
```

Une telle classe peut avoir le même contenu qu'une classe normale. Ses enfants pourront utiliser tous ses éléments déclarés (attributs et méthodes). Cependant, ce type de classe permet de définir des méthodes abstraites. Ces méthodes ont une particularité ; elles n'ont pas de corps !

Créer la classe Abstraite Animal contenant la méthodes abstraite `manger()`

```
1  abstract class Animal{  
2      abstract void manger();//une méthode abstraite  
3  }
```

Pour une "méthode abstraite", il est difficile de voir ce que cette méthode sait faire...

Retenez bien qu'une méthode abstraite n'est composée que de l'entête de la méthode suivie d'un point-virgule : **;**

Créer la méthode abstraite `manger ()` avec un corps (**{}**)

Formation Orienté Objet et Java de base

Atelier 6 : Abstract

Encadré par M.BOULCHAOUB

Vous devez savoir qu'une méthode abstraite **ne peut exister** que dans une classe abstraite. Si dans une classe, vous avez une méthode déclarée abstraite, vous **DEVEZ DÉCLARER CETTE CLASSE COMME ETANT ABSTRAITE.**

Essayer de déclarer une méthode abstraite dans une classe concrète.

Dans ce cas, nos classes enfants hériteront aussi des classes abstraites mais, vu que celles-ci n'ont pas de corps, nos classes enfants seront **OBLIGÉES de redéfinir ces méthodes !**

Créer une classe fille d'Animal et nommer la Loup.

Créer une classe fille d'Animal et nommer la Chien.

Dans la classe de Test, Déclarer un premier objet en utilisant Animal et instancier en utilisant Loup.

Dans la classe de Test, Déclarer un deuxième objet en utilisant Animal et instancier en utilisant Chien.

Vous aurez alors :

```
1  public class Test{
2
3      public static void main(String args[]){
4
5          Animal loup = new Loup();
6          Animal chien = new Chien();
7          loup.manger();
8          chien.crier();
9
10     }
11 }
```

On sait que les classes abstraites ne peuvent pas être instanciées?

Ici, nous n'avons pas instancié notre classe abstraite. Nous avons instancié un objet Loup que nous avons mis dans un objet de type Animal : il en va de même pour l'instanciation de la classe Chien.

Vous devez vous rappeler que l'instance se crée avec le mot clé **new**. En aucun cas le fait de déclarer une variable d'un type de classe donnée (ici, Animal) est une instanciation ! Ici nous instancions un Loup et un Chien.

Formation Orienté Objet et Java de base

Atelier 6 : Abstract

Encadré par M.BOULCHAOUB

Vous pouvez aussi utiliser une variable de type **Object** comme référence pour un objet **Loup**, un objet **Chien**...

Essayer avec ce code:

```
1 public class Test{
2     public static void main(String[] args){
3         Object obj = new Loup();
4         Loup l = obj; //Problème de référence
5     }
6 }
```

Eh oui ! Vous essayez ici de mettre une référence de type *Object* dans une référence de type *Loup* Pour avertir la JVM que la référence que vous voulez mettre dans votre objet de type *Loup* est un *Loup*, vous devez utiliser le *transtypage* !

```
public class Test{
    public static void main(String[] args){
        Object obj = new Loup();
        Loup l = (Loup)obj; //Vous prévenez la JVM que la référence que vous passez est de type Loup
    }
}
```

Vous pourrez bien évidemment instancier directement un objet *Loup*, un objet *Chien* ou tout autre.

Pour le moment, nous n'avons aucun code dans aucune classe ! Les exemples fournis ne font rien du tout, mais ils seront censés fonctionner lorsque nous aurons mis des morceaux de code dans nos classes.

À présent, étoffons nos classes et notre diagramme avant d'avoir un code qui fonctionne bien !

Étoffons notre exemple

Nous allons donc rajouter des morceaux de code à nos classes.
Tout d'abord, voyons le topo de ce que nous savons.

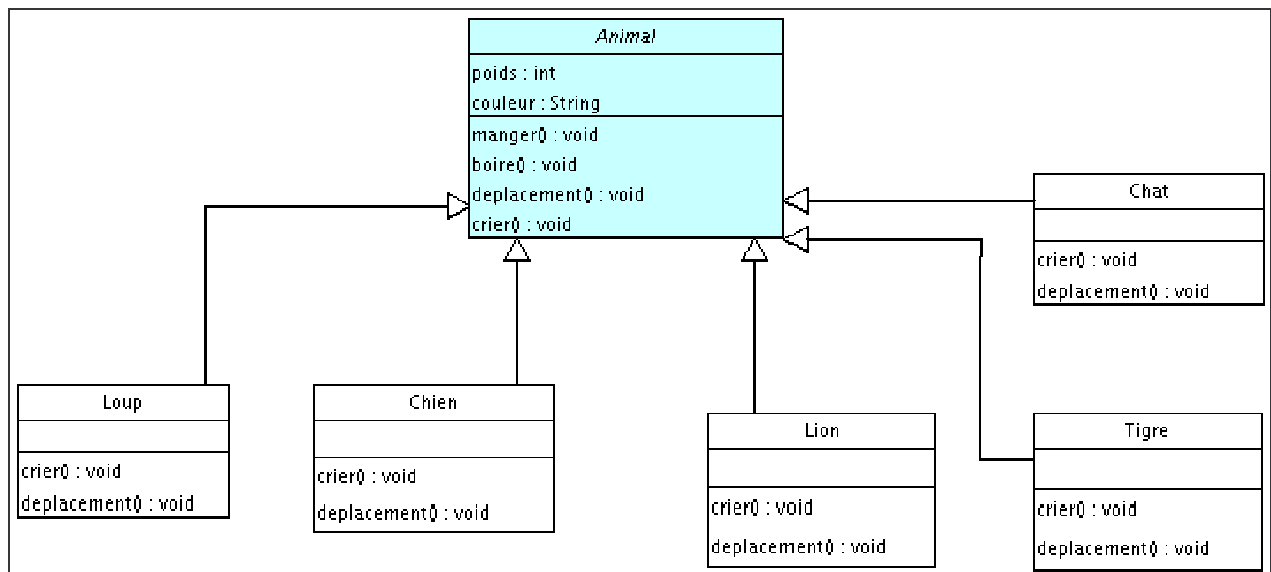
Formation Orienté Objet et Java de base

Atelier 6 : Abstract

Encadré par M.BOULCHAHOU

- Nos objets auront tous une couleur et un poids différents. Nos classes auront donc le droit de modifier ceux-ci
- Ici, nous partions du principe que tous nos animaux mangeront de la viande. La méthode **manger ()** sera donc définie dans la classe *Animal*
- Idem pour la méthode **boire ()**. Ils boiront tous de l'eau.
- Par contre, ils ne crient pas et ne se déplaceront pas de la même manière. Nous déclarerons les méthodes **crier ()** et **déplacement ()** abstraites dans la classe *Animal*.

Voici ce que donneraient nos classes :



La classe abstraite en bleu.

Nous voyons bien que notre classe **Animal** est déclarée abstraite et que nos classes filles héritent de celle-ci. De plus, nos classes filles ne redéfinissent que deux méthodes sur quatre, on en conclut ici que ces deux méthodes doivent être abstraites.

Nous ajouterons deux constructeurs à nos classes filles, un par défaut, ainsi qu'un avec les deux paramètres d'initialisation. À ceci nous ajouterons aussi les

Formation Orienté Objet et Java de base

Atelier 6 : Abstract

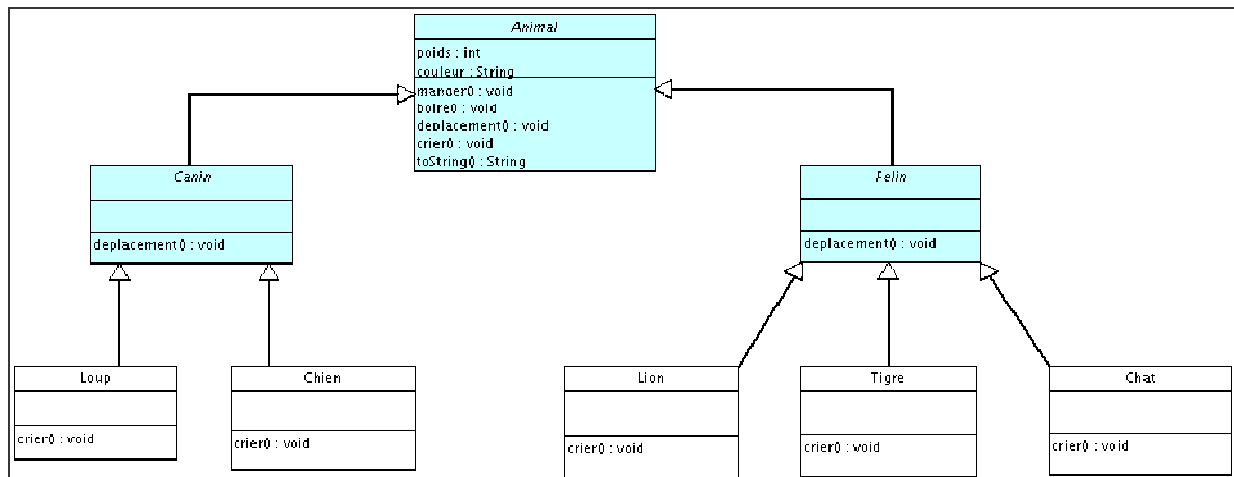
Encadré par M.BOULCHAOUB

accesseurs d'usage. Cependant... nous pouvons améliorer un peu cette architecture, sans pour autant rentrer dans les détails !

Vu les animaux présents, nous aurions pu faire une sous-classe *Carnivore*, ou encore *AnimalDomestique* et *AnimalSauvage*... Ici, nous allons nous contenter de faire deux sous-classes **Canin** et **Felin** qui hériteront d'*Animal* et dont nos objets hériteront !

Nous allons redéfinir la méthode déplacement () dans cette classe car nous allons partir du principe que les félins se déplacent d'une certaine façon, et les canins d'une autre. Avec cet exemple, nous réviserons le polymorphisme...

Voilà notre diagramme mis à jour :



Vous avez vu ? J'ai ajouté une méthode toString() à la classe *Animal*.

Voici les codes Java correspondant :

Formation Orienté Objet et Java de base

Atelier 6 : Abstract

Encadré par M.BOULCHAHOU

Animal.java

```
1  abstract class Animal {
2
3      /**
4       * La couleur de l'animal
5       */
6      protected String couleur;
7      /**
8       * Le poids
9       */
10     protected int poids;
11
12     /**
13      * La méthode manger
14      */
15     protected void manger() {
16         System.out.println("Je mange de la viande");
17     }
18
19     /**
20      * La méthode boire
21      */
22     protected void boire() {
23         System.out.println("Je bois de l'eau !");
24     }
25
26     /**
27      * La méthode de déplacement
28      */
29     abstract void déplacement();
30     /**
31      * La méthode de cri
32      */
33     abstract void crier();
34 }
```

Vous devez rajouter la l'implémentation de la méthode toString () ;

Felin.java

```
1  public abstract class Felin extends Animal {
2
3      @Override
4      void déplacement() {
5          System.out.println("Je me déplace seul !");
6      }
7
8  }
```

Canin.java

Formation Orienté Objet et Java de base

Atelier 6 : Abstract

Encadré par M.BOULCHAHOU

```
1  public abstract class Canin extends Animal {
2
3      @Override
4      void deplacement() {
5          System.out.println("Je me déplace en meute !");
6      }
7
8  }
```

Chien.java

```
1  public class Chien extends Canin {
2
3      public Chien() {
4
5      }
6      public Chien(String couleur, int poids){
7          this.couleur = couleur;
8          this.poids = poids;
9      }
10
11
12      void crier() {
13          System.out.println("J'aboie sans raison ! ");
14      }
15
16  }
```

Loup.java

```
1  public class Loup extends Canin {
2
3      public Loup() {
4
5      }
6      public Loup(String couleur, int poids){
7          this.couleur = couleur;
8          this.poids = poids;
9      }
10
11      void crier() {
12          System.out.println("J'hurle à la lune en faisant ouhouh ! ! ");
13      }
14  }
```

Lion.java

Formation Orienté Objet et Java de base

Atelier 6 : Abstract

Encadré par M.BOULCHAHOU

```
1  public class Lion extends Felin {
2
3      public Lion() {
4
5      }
6      public Lion(String couleur, int poids) {
7          this.couleur = couleur;
8          this.poids = poids;
9      }
10
11     void crier() {
12         System.out.println("Je rugis dans la savane !");
13     }
14
15 }
```

Tigre.java

```
1  public class Tigre extends Felin {
2
3      public Tigre() {
4
5      }
6      public Tigre(String couleur, int poids) {
7          this.couleur = couleur;
8          this.poids = poids;
9      }
10
11     void crier() {
12         System.out.println("Je grogne très fort !");
13     }
14
15 }
```

Chat.java

Formation Orienté Objet et Java de base

Atelier 6 : Abstract

Encadré par M.BOULCHAHOU

```
1  public class Chat extends Felin {
2
3
4      public Chat() {
5
6      }
7      public Chat(String couleur, int poids) {
8          this.couleur = couleur;
9          this.poids = poids;
10     }
11
12     void crier() {
13         System.out.println("Je miaule sur les toits !");
14     }
15
16 }
```

Dis donc ! Une classe abstraite ne doit pas avoir une méthode abstraite ?

*Je n'ai jamais dit ça ! Une classe déclarée abstraite n'est plus **instanciable**, mais elle **n'est nullement obligée** d'avoir des méthodes abstraites. En revanche, une classe ayant une méthode abstraite doit être déclarée abstraite.*

Maintenant que vous avez toutes vos classes. Faites des tests. Autant que vous le voulez.

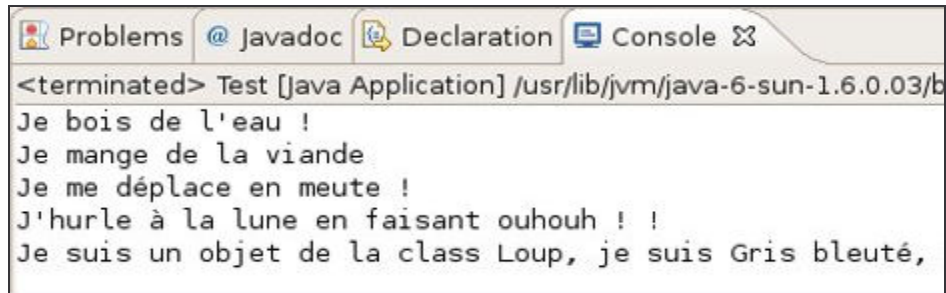
```
1  public class Test {
2
3      /**
4       * @param args
5       */
6      public static void main(String[] args) {
7          Loup l = new Loup("Gris bleuté", 20);
8          l.boire();
9          l.manger();
10         l.deplacement();
11         l.crier();
12         System.out.println(l.toString());
13     }
14
15 }
```

Voilà le jeu d'essai de ce code :

Formation Orienté Objet et Java de base

Atelier 6 : Abstract

Encadré par M.BOULCHAHOU



```
<terminated> Test [Java Application] /usr/lib/jvm/java-6-sun-1.6.0.03/b
Je bois de l'eau !
Je mange de la viande
Je me déplace en meute !
J'hurle à la lune en faisant ouhouh ! !
Je suis un objet de la class Loup, je suis Gris bleuté,
```

Dans cet exemple, nous pouvons voir que nous avons un objet Loup.

- À l'appel de la méthode **boire** () : l'objet appelle la méthode de la classe Animal.
- À l'appel de la méthode **manger** () : idem.
- À l'appel de la méthode **toString** () : idem.
- À l'appel de la méthode **déplacement** () : c'est la méthode de la classe **Canin** qui est invoquée ici.
- À l'appel de la méthode **crier** () : c'est la méthode de la classe Loup qui est appelée.

Remplacez le type de référence (ici, Loup) par Animal ou Object. Essayez avec des objets Chien, etc. Et vous verrez que tout fonctionne, excepté que vous ne pourrez pas instancier d'Object, de Felin ou de Canin !