Chapitre XII

Classes Abstraites et Interfaces

Note: Ce chapitre largement inspiré de https://openclassrooms.com/courses/apprenez-a-programmer-en-java/les-classes-abstraites-et-les-interfaces

Dans ce chapitre le mot *Interface* a une signification spécifique en Java. Il ne fait pas référence aux interfaces graphiques, ou interfaces Homme-machine.

1 Introduction

En Java et en POO de manière générale, les *Classes Abstraites* et les *Interfaces* vont vous permettre de mieux structurer vos programmes.

Grâce aux chapitres précédents, vous pouvez anticiper que vos programmes Java regorgeront de classes, avec de l'héritage, des dépendances, de la composition.... Afin de bien structurer vos programmes (on parle d'architecture logicielle), vous allez devoir organiser le comportement (les méthodes) des objets dans les classes. Doivent ils être dans la classe mère ? la classe fille ? Ce chapitre explique comment obtenir une structure assez souple pour pallier aux problèmes de programmation les plus courants.

2 Classes Abstraites

On utilise des les classes abstraites pour modéliser des concepts abstraits et théoriques ou des concepts non définis (non complètement déterminés / écrits / concrets).

Cela permet de spécifier des membres communs à toute une hiérarchie (c'est-à-dire toutes les classes dérivées).

Une classe abstraite permet de forcer toutes ses classes dérivées (non abstraites) à implémenter un certain nombre de comportements (méthodes)

⟼ Cela permet de garantir l'existence d'une méthode quelque soit la classe dérivée manipulée et quequ'en soit son implémentation

⟼ Cela permet d'écrire des algorithmes directement en utilisant des références sur les classes abstraites.

Une classe abstraite n'est pas instanciable :

- On ne peut pas utiliser l'opérateur `new` sur un de ses constructeurs
- On n'a jamais d'instance manipulable en mémoire

Grâce au polymorphisme, on peut utiliser dans des programmes des références à cette classe. On pourra ainsi manipuler des instances de classes dérivées de manière générique.

2.a Syntaxe

En UML, les classes abstraites ont leur nom écrit en *italique*.

En Java, on utilise le mot clé `abstract`.

```
abstract class NomDeLaClasse {
        ...
}
```

Dans une méthode `main`, on peut **déclarer** une référence sur cette classe, mais pas l'instancier:

```
NomDeLaClasse ref; // Cette ligne fonctionne
ref = new NomDeLaClasse(); // Cette ligne n'est pas possible.
```

Pour obliger l'implémentation d'une méthode, une classe abstraite peut définir une ou plusieurs méthodes abstraites c'est-à-dire définir la signature de cette méthode sans l'implémenter. Exemple:

```
abstract class NomDeLaClasse {
        public abstract void methode();
}
```

On note ici qu'il n'y a pas d'implémentation et que la signature de la méthode est suivie d'un `;`.

La déclaration d'une méthode abstraite dans une classe a plusieurs conséquences:

- la classe **doit** être abstraite
- une classe qui hérite de cette classe abstraite peut
  - soit implémenter cette méthode
  - soit ne pas implémenter cette méthode, auquel cas cette sous-classe est alors abstraite

Une classe abstraite peut avoir des méthodes abstraites, mais aussi implémenter d'autres méthodes. Cela va permettre de capitaliser du code commun à toute la hiérarchie.

2.b Exemple

Prenons un exemple simple de programme qui gère différents types d'animaux (pas forcément réaliste mais qui a le mérite d'être simple à comprendre et efficace pour l'illustration des différents concepts abordés). Dans ce programme, vous aurez des loups, des chiens, des chats, des lions et des tigres. Comme vous avez bien lu et compris le chapitre précédent, vous remarquez que tous ces animaux ont des points communs... Et qui dit point commun en POO, dit *héritage* ! Nous allons donc définir comme points communs à tous ces animaux, le fait qu'ils aient:

- une couleur
- un poids
- un cri
- une façon de se déplacer
- qu'ils mangent quelque chose
- qu'ils boivent quelque chose

Une première étape est de définir ce qui correspond aux données (*attributs*) et aux comportements (*méthodes*). Une première solution (utilisant uniquement l'héritage, pas encore de classes abstraite, c'est pourquoi il n'y a pas de nom de classe en italique) est exprimée dans le diagramme de classes de la figure XII.1:

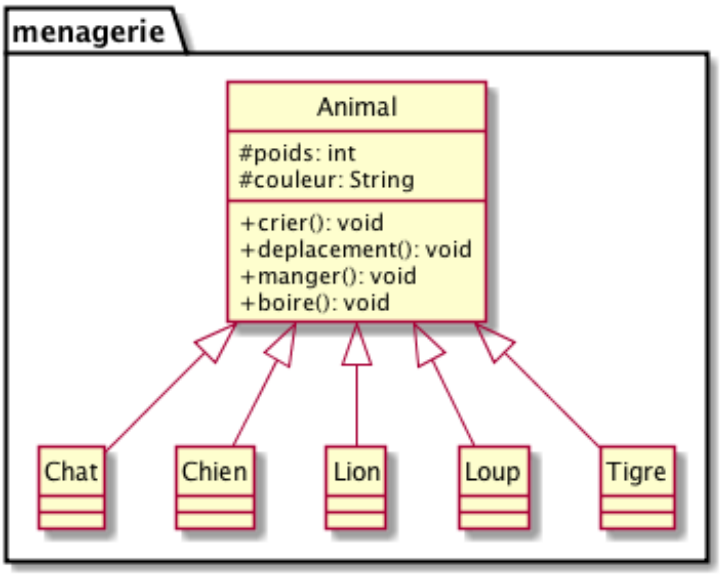

Figure XII.1: Première ébauche de la ménagerie. Une classe mère et 5 classes filles.

```
1  package menagerie;
2  public class Animal {
3      protected int poids;
4      protected String couleur;
5
6      public void crier() {}
7
8      public void deplacement() { }
9
10     public void manger() {
11         System.out.println("Je mange de la viande.");
12     }
13
14     public void boire() {
15         System.out.println("Je bois de l'eau.");
16     }
17
18 }
```

Figure XII.2: Première implémentation de la classe `Animal`. Que mettre dans les méthodes `crier()` et `deplacement()` ?

Si l'on part du principe[1] que tous les animaux présents mangent de la viande et boivent de l'eau[2], on peut obtenir l'implémentation de la classe `Animal` figure XII.2.

On sait que tous les animaux ont un cri. Mais on ne peut pas définir un cri (i.e. dans notre cas une méthode `cri`) générique pour chaque animal. Les classes `Loup`, `Chien`, `Lion`, `Tigre` et `Chat` devront chacune définir leur méthode `cri`. Si on laisse la méthode `cri` de la classe `Animal` avec des accolades vides (que mettre, de toute façon ?), on devra de toute façon redéfinir cette méthode dans les sous-classes. La méthode `cri` n'est présente dans la classe `Animal` que pour définir un comportement par défaut pour tous les animaux: *tous les animaux ont un cri*, sans pour autant implémenter ce comportement.

Plutôt que de laisser des accolades vides, on va déclarer la méthode `cri` *abstraite*, c'est-à-dire la déclarer sans la définir. Comme la classe `Animal` contiendra au moins une méthode abstraite, elle sera à

[1] Rappel: cet example n'a pas pour intérêt d'être réaliste, uniquement d'illustrer l'abstraction d'une classe...
[2] ...et puis dans le cas de *nos* animaux, c'est vrai...

son tour *abstraite*.

Si l'on observe les animaux présents, nous aurions pu faire une sous-classe `Carnivore`, ou encore `AnimalDomestique` et `AnimalSauvage`... Dans cet exemple, nous considérerons plutôt les caractéristiques de déplacement pour les sous-classes. Ici, on fait l'hypothèse que les animaux *canins* se déplacent en meute, alors que les *félins* se déplacent seuls.

On obtient le diagramme de classes de la figure XII.3 avec les implémentations XII.4. L'implémentation de la classe `Tigre` est laissée ne exercice (elle ressemble trait pour trait à celle de la classe `Lion`).

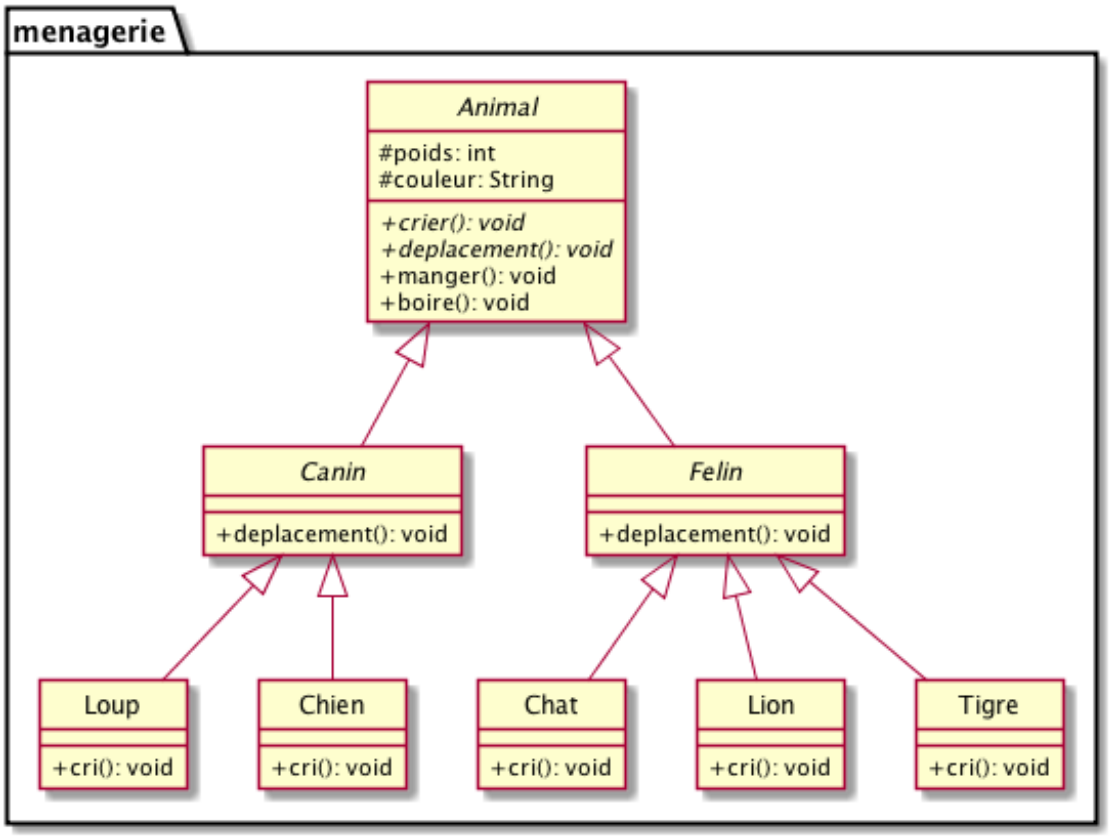

Figure XII.3: Une ménagerie plus aboutie.

```java
package menagerie;

public abstract class Canin
    extends Animal {
    @Override
    public void deplacement() {
        System.out.println("Je me
            déplace en meute.");
    }
}
```

Fichier `Canin.java`.

```java
package menagerie;

public abstract class Felin
    extends Animal {
    @Override
    public void deplacement() {
        System.out.println("Je me
            déplace seul.");
    }
}
```

Fichier `Felin.java`.

```java
package menagerie;
public abstract class Animal {
    protected int poids;
    protected String couleur;

    public abstract void crier();

    public abstract void deplacement();

    public void manger() {
        System.out.println("Je mange de la viande.");
    }

    public void boire() {
        System.out.println("Je bois de l'eau.");
    }
}
```

Figure XII.4: Fichier `Animal.java`.

```java
package menagerie;

public class Loup extends Canin{

    public Loup(String couleur,
        int poids) {
        this.couleur = couleur;
        this.poids = poids;
    }

    @Override
    public void crier() {
        System.out.println("Je
            hurle à la lune en
            faisant ouhouh !");
    }

}
```

Fichier `Loup.java`.

```java
package menagerie;

public class Chien extends Canin{
    public Chien(String couleur,
        int poids) {
        this.couleur = couleur;
        this.poids = poids;
    }

    @Override
    public void crier() {
        System.out.println("J'
            aboie sans raison !");
    }

}
```

Fichier `Chien.java`.

```java
package menagerie;

public class Lion extends Felin {
    public Lion(String couleur, int
        poids) {
        this.couleur = couleur;
        this.poids = poids;
    }

    @Override
    public void crier() {
        System.out.println("Je rugis
            dans la savane !");
    }

}
```

Fichier `Lion.java`.

```java
package menagerie;

public class Chat extends Felin{
    public Chat(String couleur, int
        poids) {
        this.couleur = couleur;
        this.poids = poids;
    }

    @Override
    public void crier() {
        System.out.println("Je miaule
            sur les toits !");
    }

}
```

Fichier `Chat.java`.

Dans cet example, on peut noter les points suivants:

1. La classe `Animal` est *abstraite* et contient 2 méthodes `abstraites`: `crier(): void` et `deplacement(): void`.

2. Les classes `Canin` et `Felin` sont *abstraites*. En effet, bien qu'elles implémentent l'une des deux méthodes *abstraites* de la classe `Animal` (la méthode `deplacement(): void`), elles n'implémentent pas la méthode `crier(): void`, or par héritage, elles ont une méthode **crier(): void**. Elles doivent donc soit l'implémenter, soit être elles-même *abstraites*. C'est le deuxième cas qui se produit ici.

3. Les classes instanciables: `Loup`, `Chien`, `Lion`, `Tigre` et `Chat` on un constructeur qui prend 2 paramètres et initialise les attributs d'`Animal` qui sont aussi leurs attributs puisqu'ils sont *protected*. En fait, même si la classe `Animal` n'est pas instanciable, on aurait pu factoriser ce code dans un constructeur d'`Animal`. En effet, même si ce constructeur ne pourrait jamais être appelé avec l'opérateur `new` (rappel: la classe `Animal` **ne** peut **pas** être **instanciée**), on aurait pu l'appeler dans les classes `Loup`, `Chien`, `Lion`, `Tigre` et `Chat` à l'aide du mot clé `super`.

3 Interface

Les interfaces permettent de définir exclusivement une liste de comportements abstraits. Une interface est une classe purement abstraite qui n'a pas d'attribut et où aucune des méthodes n'est implémentée.

Définir une interface correspond à écrire un contrat avec les classes qui *implémenterons* cette interface en définissant une liste de méthodes.

Une classe qui implémente une interface est une classe qui rempli le contrat, c'est-à-dire qui implémente toutes les méthodes définies dans l'interface. L'un des avantages des interfaces est qu'elles sont déconnectées de la hiérarchie d'héritage.

3.a Syntaxe

En UML, le nom des interfaces est écrit en *italique* et précédé du mot clé «`interface`».

En Java, on utilise le mot clé `interface`.

```
1 public interface NomDeLInterface {
2         void methodeAImplementerAilleurs();
3 }
```

On peut noter que les méthodes d'une interface sont systématiquement `public`, c'est pourquoi le modificateur d'accès `public` peut être omis devant le nom des méthodes.

Une classe qui *implémente* une interface utilise le mot clé `implements`.

```
1 public class MaClasseAMoi implements NomDeLInterface {
2         void methodeAImplementerAilleurs) {
3                 // penser à écrire du code....
4                 ...
5         }
6 }
```

3.b Example

Reprenons l'exemple précédent. Imaginons qu'un développer externe au projet souhaite réutiliser nos classes (c'est la puissance de Java), dans un autre cadre. Il souhaite définir un comportement d'animal domestique: *passer par la chatière, donner la pâte*, etc.

Ces méthodes ne peuvent pas être implémentées dans la classe `Animal` puisque l'on ne souhaite pas qu'un `Loup` puisse passer par une chatière, ni qu'un `Tigre` donne la pâte... Cependant, ce comportement étant générique, on souhaite pouvoir appliquer les méthodes `passerChatiere(): void` et `donnerLaPate(): void` sans se soucier d'avoir affaire à un `Chat` ou à un `Chien`. Ici, il n'y a pas non plus de code/implémentation en commun, il s'agit seulement d'un *comportement*, une sorte de *contrat* que les animaux passent avec leur maître...

Nous allons donc utiliser l'*interface* `Domestique`. Nous obtenons la hiérarchie présentée figure XII.5 et le code XII.6.

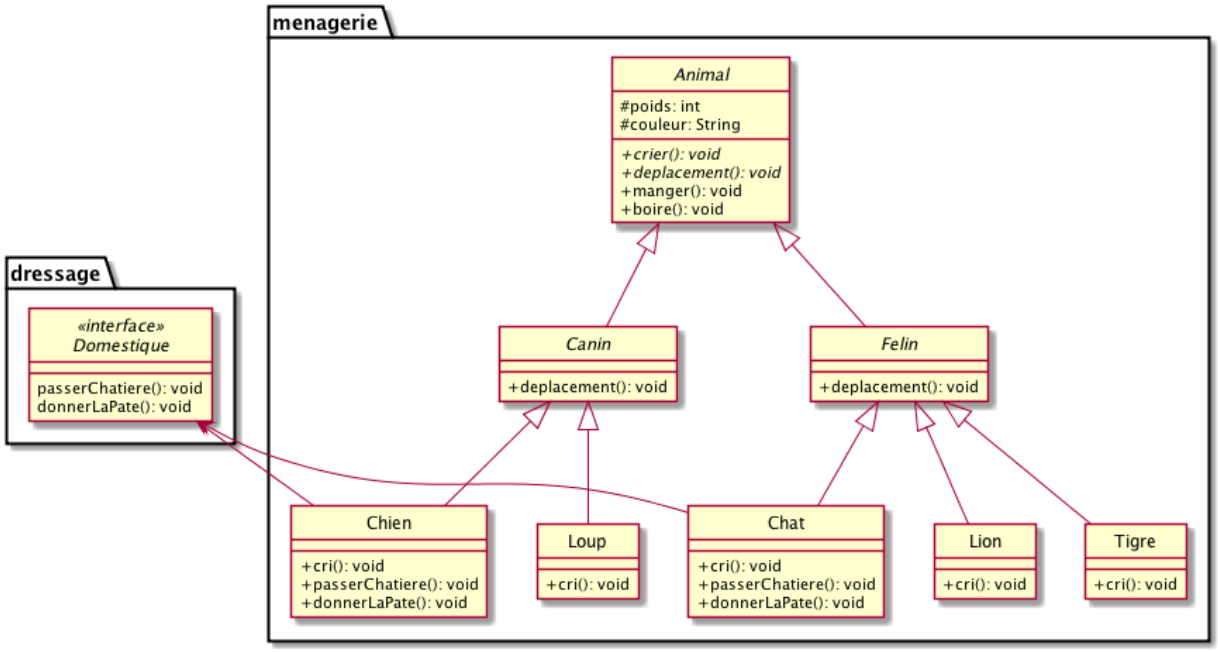

Figure XII.5: Menagerie et dressage.

```java
package dressage;

public interface Domestique {
    void passerChatiere();
    void donnerLaPate();
}
```

Figure XII.6: Fichier `Domestique.java`.

```java
package menagerie;
import dressage.Domestique;

public class Chien extends Canin
    implements Domestique{
    public Chien(String couleur, int
        poids) {
        this.couleur = couleur;
        this.poids = poids;
    }

    @Override
    public void crier() {
        System.out.println("J'aboie
            sans raison !");
    }

    @Override
    public void passerChatiere() {
        System.out.println("Un peu
            serré, mais ça passe...");
    }

    @Override
    public void donnerLaPate() {
        System.out.println("J'adô
            ooore donner la papate !")
            ;
    }

}
```

Fichier `Chien.java`

```java
package menagerie;
import dressage.Domestique;

public class Chat extends Felin
    implements Domestique{
    public Chat(String couleur, int
        poids) {
        this.couleur = couleur;
        this.poids = poids;
    }

    @Override
    public void crier() {
        System.out.println("Je miaule
            sur les toits !");
    }

    @Override
    public void passerChatiere() {
        System.out.println("Et zou,
            je rentre à la maison !");
    }

    @Override
    public void donnerLaPate() {
        System.out.println("Non, j'
            aime pas ça !");
    }

}
```

Fichier `Chat.java`.

Dans cet example, on a placé l'*interface* `Domestique` dans un autre *package*. Ce n'est pas du tout obligatoire (bien que ce soit souvent le cas). Notez l'ordre dans la déclaration des classes `Chat` et `Chien`: d'abord `extends` puis `implements`.

Les *interfaces* étant déconnectées de la hiérarchie d'héritage, une classe peut implémenter plusieurs *interfaces* (alors qu'elle peut hériter d'au plus une classe).