Chapitre 2-6
Les interfaces

1. Qu'est-ce qu'une interface ?

En Kotlin, une interface est ce qu'on appelle un contrat. Elle permet d'imposer un comportement à respecter à travers un ensemble de méthodes définies uniquement par leurs signatures. Ces méthodes devront être implémentées par la classe qui implémentera l'interface.

À quoi ça sert ?

Quand on débute en programmation et plus particulièrement en programmation orientée objet, comprendre l'intérêt des interfaces n'est pas toujours évident et leur utilisation n'est pas toujours naturelle.

Illustrons l'intérêt des interfaces en reprenant l'exemple des animaux. Jusqu'à maintenant, nous avons manipulé trois classes, à savoir une classe abstraite représentant un animal, une classe représentant un chat et héritant de la classe représentant un animal et une classe représentant un chien et héritant également de la classe représentant un animal.

Les deux animaux représentés ici ont un point commun : il s'agit d'animaux domestiques.

Ajoutons à présent de nouveaux comportements à travers de nouvelles méthodes : lécher, se blottir. Pour des raisons pédagogiques, nous souhaitons que les animaux domestiques implémentent systématiquement ces nouvelles méthodes. Dans ce cas, pourquoi ne pas les déclarer en tant que méthodes abstraites au sein de la classe `Animal` :

```
abstract class Animal(var age: Int, var name: String, var race:
String, var color: String, var size: Int, var weight: Float)
{

//...

 abstract fun lick()

 abstract fun snuggle()

 //...

}
```

Il restera à implémenter ces deux méthodes dans les classes `Cat` et `Dog` (voir le chapitre L'héritage).

En attendant, ajoutons dans le programme un nouvel animal comme un panda, un tigre, un lion ou encore un puma. Bref, un animal qui n'est pas domestique et pour lequel lécher et se blottir ne font pas forcément sens. Pour ce faire, nous pouvons introduire deux nouvelles classes abstraites. Une première représentant un animal sauvage et une seconde représentant un animal domestique. Ces deux classes seraient des classes filles de la classe `Animal`. Les classes `Cat` et `Dog` hériteraient alors de la classe représentant un animal domestique tandis que la classe `Lion` hériterait de la classe représentant un animal sauvage. C'est effectivement une possibilité.

Réfléchissons maintenant à l'introduction de nouveaux animaux, comme une tortue. Cette évolution pose pas mal de questions. Est-ce un animal domestique sauvage ? Est-ce qu'une tortue lèche et se blottit ? Disons qu'une tortue est un animal domestique pour lequel des actions comme lécher et se blottir ne font pas forcément sens.

Nous sommes dans une impasse. Nous pouvons toujours imaginer une réorganisation des classes, avec une classe abstraite représentant un animal, une classe abstraite représentant un animal domestique (avec les méthodes permettant de lécher et se blottir) n'ayant pas de lien d'héritage avec la classe `Animal`, et une classe abstraite représentant un animal sauvage n'ayant pas non plus de lien d'héritage avec la classe `Animal`. Nous pouvons envisager des combinaisons pour modéliser les classes concrètes représentant des animaux ; la classe `Cat` hériterait de la classe `Animal` et de la classe représentant un animal domestique ; la classe `Dog`, quant à elle, hériterait de la classe `Animal` et de la classe représentant un animal domestique ; la classe `Lion` hériterait de la classe `Animal` et de la classe représentant un animal sauvage.

Malheureusement, cette modélisation est impossible ! En effet, en Kotlin, l'héritage multiple n'existe pas !

Remarque

Quand on parle ici d'héritage multiple, on parle bien d'une classe (fille) qui hérite de plusieurs classes (mères), et non d'une classe qui hérite d'une classe qui hérite d'une classe, etc. (héritage sur plusieurs niveaux).

Dans ce cas, la solution consiste à utiliser des interfaces. Ainsi, tous les animaux pourront hériter de la classe `Animal` tout en ayant les comportements spécifiques que nous aurons injectés, selon qu'il s'agit d'animaux domestiques ou sauvages. Il suffit de créer deux interfaces : une première représentant les comportements des animaux domestiques et une seconde représentant les comportements des animaux sauvages.

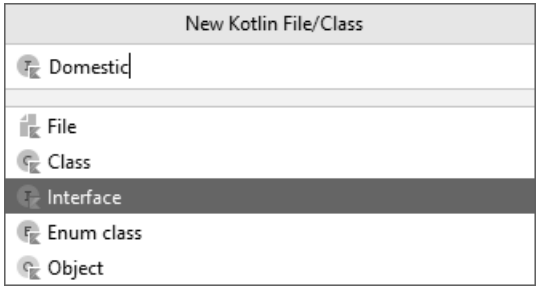
Remarque

Chez les débutants, il n'est pas toujours aisé de faire la différence entre le cas d'usage d'une interface et le cas d'usage d'un héritage.

2. Écrire une interface

Écrivons l'interface permettant d'imposer un comportement à un animal domestique.

- Pour créer cette première interface, rendez-vous dans **IntelliJ IDEA**.
- Cliquez du bouton droit sur le dossier src du programme, puis sur **New** et finalement sur **Kotlin File/Class**.
- Dans la boîte de dialogue qui s'affiche, saisissez le nom de l'interface, ici **Domestic**.
- Puis sélectionnez **Interface** comme type de fichier.
- Pour finaliser la création de ce nouveau fichier, appuyez sur la touche [Entrée] du clavier.



Un fichier Domestic.kt apparaît alors dans le dossier src du programme. Son contenu est le suivant :

```
interface Domestic
{
}
```

C'est le code minimal pour créer une interface. Pour déclarer une interface, il convient donc d'utiliser le mot-clé `interface`, suivi du nom de l'interface. Dans ce cas : `Domestic`.

Remarque

Comme pour une classe, la convention de nommage en Kotlin stipule que le nom d'une interface débute toujours par une majuscule et respecte la notation CamelCase.

Pour le moment il n'y a pas de code entre les accolades de cette interface, mais c'est entre celles-ci que nous allons écrire les comportements spécifiques aux animaux domestiques, à savoir lécher et se blottir.

Une interface peut, dans un cadre pédagogique, être considérée comme une classe 100 % abstraite. Cela signifie que lorsque l'on écrit les méthodes de l'interface, on doit, comme dans le cas des méthodes abstraites, écrire uniquement leurs en-têtes, à savoir le nom, les éventuels paramètres et un éventuel type de retour.

Remarque

Bien qu'on puisse considérer une interface comme une classe 100 % abstraite, l'utilisation du mot-clé `abstract` n'est pas attendue. Le droit d'accès d'une méthode présente dans une interface est obligatoirement `public`.

Comme annoncé, reportons les méthodes `lick` et `snuggle` de la classe `Animal` dans l'interface `Domestic` :

```
interface Domestic
{

 fun lick()

 fun snuggle()

}
```

3. Implémenter une interface

Maintenant que l'interface est en place, il convient de la relier aux classes `Cat` et `Dog` qui représentent toutes les deux des animaux domestiques.

Pour traduire le fait que la classe `Cat` implémente l'interface `Domestic`, on écrit, avant l'accolade ouvrante de la classe, le symbole `:` suivi du nom de l'interface que l'on souhaite implémenter, dans ce cas l'interface `Domestic` :

```
class Cat(age: Int, name: String, race: String, color: String,
size: Int, weight: Float)
 : Domestic
{


 //...

}
```

La syntaxe est très proche de celle permettant d'exprimer l'héritage. À ce propos, dans cet exemple, la classe `Cat` n'hérite plus de la classe `Animal`. Corrigeons tout de suite ce problème.

En Kotlin, pour exprimer le fait qu'une classe hérite d'une autre classe et implémente une interface, on utilise le symbole `:` suivi du nom de la classe dont on hérite (ici `Animal`) et du constructeur de classe mère à appeler. Vient ensuite le symbole `,` suivi du nom de l'interface à implémenter (ici `Domestic`).

```
class Cat(age: Int, name: String, race: String, color: String,
size: Int, weight: Float)
 : Animal(age, name, race, color, size, weight),
   Domestic
{


 //...

}
```

Remarque

Si l'héritage multiple n'est pas possible en Kotlin, il est possible pour une classe d'implémenter plusieurs interfaces. Dans ce cas, il convient simplement d'écrire le nom de toutes les interfaces à implémenter en les séparant par une virgule (,).

Pour le moment, si nous avons exprimé le fait que la classe `Cat` implémente l'interface `Domestic`, nous n'avons pas encore finalisé cette implémentation. Pour ce faire, il convient simplement d'écrire le corps des méthodes définies dans l'interface, exactement de la même manière que pour des classes avec des méthodes abstraites (à l'aide du mot-clé `override`).

La classe `Cat` devient alors :

```
class Cat(age: Int, name: String, race: String, color: String,
size: Int, weight: Float)
 : Animal(age, name, race, color, size, weight),
   Domestic
{

 fun meow()
 {
   println("meow! Meow!")
 }

 override fun eat(foodWeight: Int)
 {
   weight += (foodWeight / 10000f)
 }

 override fun lick()
 {
   println("lick, lick")
 }

 override fun snuggle()
 {
   println("snuggle")
 }

}
```