Contrairement à d'autres langages de POO, la notion d'héritage multiple n'existe pas en Java : une classe fille ne peut pas être dérivée de plusieurs classes parentes. Pour pallier cette limitation, le langage Java a introduit la notion d'interface.

11. Interfaces

Une interface est un moyen de créer un contrat pour les classes et donc les objets, contrat qui permettra de définir des comportements obligatoires pour toutes les classes **implémentant** cette interface.

L'intérêt principal d'une interface est de factoriser des comportements communs pour une utilisation standardisée. Si, au cours du cycle de développement de votre application, vous vous apercevez qu'un de vos objets peut avoir une meilleure implémentation (comme une voiture avec un meilleur moteur), et si vous utilisez une interface pour communiquer avec cet objet, vous pourrez modifier votre code beaucoup plus facilement en substituant votre ancienne classe avec la nouvelle qui possédera exactement la même interface.

Reprenons l'exemple de la voiture : chaque modèle de voiture est unique. Chaque modèle est construit suivant un schéma différent, avec des technologies différentes. Néanmoins, l'interface de chaque modèle de voiture est toujours plus ou moins la même : un volant, des pédales pour le frein et l'accélérateur, les commandes de clignotants... Le volant est toujours devant vous, les pédales sont à la même place quelle que soit la voiture que vous utilisez, comme les commandes de clignotants... Félicitations ! Vous connaissez bien l'interface pour conduire une voiture et n'avez pas à repasser votre permis de conduire à chaque fois que vous vous asseyez dans une voiture différente de la vôtre, comme une voiture anglaise.

Vous avez donc déjà appris à maîtriser une interface et pas une implémentation particulière, et ce dans le monde réel. Il est possible de transposer cette maîtrise dans une application Java à l'aide d'interfaces que l'on déclare comme ceci :

```
interface Animal {
    void manger(Object deLaNourriture) ;
}
```

© Editions ENI - All rights reserved

Une interface ne peut habituellement contenir que des signatures de méthodes publiques et abstraites, donc sans aucune implémentation (pas de crochets), et des propriétés publiques, statiques ou finales.

Le fait que le concept d'héritage multiple n'existe pas en Java a déjà été souligné. Chat ne peut ainsi hériter simultanément des comportements et caractéristiques de plusieurs classes.

Par contre, Chat peut implémenter autant d'interfaces que nécessaire.

```
public final class Chat extends Felin implements Animal,
                                                  EtreVivant, Mobile {
}
```

L'interface peut être étendue (dérivée) par d'autres interfaces au moyen du mot-clé **extends**, comme avec une classe. La différence majeure avec une classe réside dans le fait qu'une interface peut elle-même hériter de plusieurs autres interfaces. Les interfaces permettent en outre de structurer le code de manière plus sûre. En effet, lorsqu'une classe implémente une interface avec le mot-clé **implements**, elle doit obligatoirement redéfinir toutes les méthodes de celle-ci. Ceci peut paraître contraignant mais l'application a ainsi l'assurance que les services ou contrats attendus de l'interface implémentée seront respectés.

```
interface Animal extends EtreVivant, Mobile {
    void manger(Object deLaNourriture) ;
}
```

Ici, toute classe voulant implémenter l'interface Animal devra obligatoirement créer une méthode publique **void manger(Object deLaNourriture)**, en plus des méthodes présentes dans les interfaces EtreVivant et Mobile.

Le mécanisme des interfaces est très utilisé à l'intérieur de Java lui-même. Il y permet de créer de la modularité et des facilités de substitution de code. Et Java permet d'utiliser et de créer cette modularité et cette facilité dans **votre** application !

Remarque

Il est conseillé d'éviter de créer des interfaces avec trop de méthodes et donc de contrats. Cela nuit globalement à la modélisation et au cycle de vie de l'application.

© Editions ENI - All rights reserved

Enfin, Java 8 permet de définir des implémentations par défaut dans les interfaces, à l'aide du mot-clé **default**.

```
interface Animal {
    default void manger(Object deLaNourriture) {
            System.out.println(this + " mange " + deLaNourriture);
    }
}
```

Si une méthode d'une interface a une implémentation par défaut, il n'est pas nécessaire d'implémenter cette méthode : l'implémentation par défaut est utilisée.

12. Polymorphisme

Le polymorphisme est une notion importante en POO, et veut dire étymologiquement « qui peut prendre plusieurs formes ».

Le polymorphisme est la capacité d'un code à choisir dynamiquement à l'exécution la méthode à exécuter, et implique qu'un même appel de méthode puisse avoir des comportements différents, soit en fonction des paramètres de cet appel, soit en fonction de l'objet avec lequel l'appel est effectué.

Java comporte deux types de polymorphismes principaux : la surcharge et la redéfinition.

Le mécanisme de polymorphisme peut également être obtenu à l'aide d'interfaces.

Depuis Java 1.5, il existe également un polymorphisme dit paramétrique, avec l'utilisation de génériques, qui sera étudié au chapitre suivant.

12.1 Par surcharge

Le polymorphisme de **surcharge** correspond à la capacité d'un objet à choisir au moment de l'exécution (dynamiquement) la méthode qui correspond le mieux à ses besoins parmi ses propres méthodes ou celles des méthodes mères.

© Editions ENI - All rights reserved

Une autre particularité de ce polymorphisme est que toutes les méthodes concernées portent le même nom. Au sein d'une classe, elles se différencient obligatoirement et uniquement par le nombre de paramètres ou le type de ces derniers.

Remarque

Rappel : une signature de méthode ne peut être implémentée qu'une seule fois dans une même classe.

La classe **Chat** est modifiée pour mettre en évidence ce mécanisme de surcharge.

```
public class Chat {
   ...
   public void manger(String nourriture){
      System.out.println("Je mange de " + nourriture);
   }

  public void manger() {
      System.out.println("Je vais aller chasser ma nourriture");
  }

   public void manger(List<Object> nourriture){
      System.out.println("Je mange tout ça : " + nourriture);
   }
}
```

Remarque

*La notation **List<Object>** définit un ensemble de données de type Object sur lequel il est possible d'itérer (parcourir tous les éléments). Ces notions seront développées dans le prochain chapitre dans les sections Collections et Génériques.*

Le programme appelant peut alors être modifié comme :

```
leChat.manger("la pâtée");
leChat.manger();
leChat.manger(Arrays.asList("croquettes", "pâté")) ;
```

© Editions ENI - All rights reserved

À l'exécution, le résultat suivant est obtenu :

```
Je mange de la pâtée
Je vais aller chasser ma nourriture
Je mange tout ça : [croquettes, pâté]
```

Ce mécanisme de surcharge peut également être appliqué aux constructeurs.

```
public class Chat {
    private Chat() {
    }

    Chat(String nom, String couleur) {
          // créer un chat naissant maintenant
          this(nom, couleur, new Date()) ;
    }
    Chat(String nom, String couleur,
                Date dateDeNaissance){
          this(nom, couleur, dateDeNaissance.getTime()) ;
    }
    Chat(String nom, String couleur,
                long dateDeNaissance) {
          this() ;
          ...
    }
}
```

12.2 Par redéfinition

Les méthodes de la classe mère peuvent être **redéfinies** dans les classes filles ou sous-classes par une opération de sous-typage. Il s'agit en fait d'une **spécialisation**. Dans le sens inverse, c'est une généralisation qui permet de factoriser les comportements communs.

D'un point de vue pratique, les méthodes des classes filles sont réécrites afin d'obtenir le comportement voulu. À l'utilisation, on a l'impression d'appeler la méthode de la classe mère, mais en fait le code de la classe fille est exécuté.

© Editions ENI - All rights reserved

Un exemple classique est la méthode **toString()** de la classe **Object**, présente dans TOUTES les classes de l'application. Elle permet de fournir une description textuelle de l'objet instancié. Cette méthode peut être redéfinie dans les classes de l'application pour fournir plus d'informations contextuelles sur les objets.

En reprenant le cas des animaux, tous les mammifères se nourrissent mais les chats domestiques mangent des croquettes, ce qui n'est pas forcément le cas des tigres.

Avec les classes déjà créées, cela donne :

```
public class Felin {
  ...
  public void manger(Object nourriture){
  }
}

public class Chat extends Felin {
  ...
  @Override
  public void manger(Object nourriture){
    System.out.println("Je mange des " + nourriture);
  }
}
public class Tigre extends Felin {
  ...
  @Override
  public void manger(Object nourriture){
    if( "croquettes".equals(nourriture)) {
          System.out.println("Un tigre ne mange pas de ça");
    }
    ...
  }
}
```

Remarque

*La ligne **@Override** est une **annotation**, notion qui sera développée dans le prochain chapitre. Elle signifie ici que la méthode annotée (immédiatement en dessous) est obligatoirement redéfinie et lèvera une erreur à la compilation si ce n'est pas le cas.*

© Editions ENI - All rights reserved

Exemple dans un programme :

```
leChat.manger("croquettes") ;
leTigre.manger("croquettes") ;
```

Le résultat suivant est obtenu à l'exécution :

```
Je mange des croquettes
Un tigre ne mange pas de ça
```

Pour stopper ce mécanisme de redéfinition, le mot-clé **final** peut être utilisé. Dans ce cas, les méthodes ne pourront pas être redéfinies (réécrites).

Exemple :

```
public class Felin {
   ...
  public final void manger(Object nourriture){
    System.out.println("Je mange " + nourriture);
  }
}

public class Chat extends Felin {
  ...
  // erreur à la compilation
  public void manger(Object deLaNourriture){
  }
}
```

12.3 Par interface

Le polymorphisme par interface permet d'avoir des méthodes ayant le même nom et les mêmes types (on parle de **signature identique**) dans des classes différentes (sans aucun lien d'héritage entre elles), dont l'implémentation pourra également être différente.

Afin de conserver un lien sémantique entre ces méthodes, il est important de créer une interface portant cette méthode, et d'implémenter cette interface dans les différentes classes. Cela permet d'appeler cette méthode au niveau de l'interface sans connaître a priori la classe effective qui concrétise cette méthode.

© Editions ENI - All rights reserved

Exemple de code :

```
public interface Ecouteur {
    void quelqueChoseEstArrive(Object quelqueChose) ;
}

class PremierEcouteur implements Ecouteur {
    public void quelqueChoseEstArrive(Object quelqueChose) {
          System.out.println("action 1") ;
    }
}
class SecondEcouteur implements Ecouteur {
    public void quelqueChoseEstArrive(Object quelqueChose) {
          System.out.println("action 2") ;
    }
}
```

Le résultat suivant est obtenu à l'exécution :

```
Ecouteur ecouteur = null ;

ecouteur = new PremierEcouteur() ;
ecouteur.quelqueChoseEstArrive(null) ; // action 1

ecouteur = new SecondEcouteur() ;
ecouteur.quelqueChoseEstArrive(null) ; // action 2
```

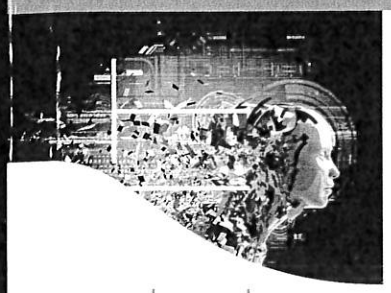
Le code ci-dessus crée des objets avec les classes concrètes **PremierEcouteur** et **SecondEcouteur**, et les stocke dans une variable de type interface **Ecouteur**. Le type concret de l'objet disparaît donc à ce moment. Pourtant, en appelant la même méthode **quelqueChoseEstArrive()**, les comportements des deux objets diffèrent.

Remarque

Ce n'est pas un polymorphisme de redéfinition, car les classes n'ont pas une hiérarchie commune. Le polymorphisme est basé ici sur le contrat que les classes implémentant l'interface doivent respecter.

© Editions ENI - All rights reserved

INFORMATIQUE
TECHNIQUE



Nouvelle édition

Expert

Java et Eclipse

Développez une application avec Java et Eclipse

Téléchargement
www.editions-eni.fr

Frédéric DÉLÉCHAMP
Henri LAUGIÉ

eni Editions