# Modélisation et programmation par Objets 1
# HLIN406

Marianne Huchard, Clémentine Nebut

1er février 2017

# Chapitre 1

# Spécialisation/généralisation et héritage

## 1.1 Généralisation - Spécialisation

Nous abordons dans ce chapitre, un des atouts majeurs de la programmation objet et qui a pour point de départ un « double » mécanisme d'inférence intellectuelle : la généralisation et la spécialisation, deux mécanismes relevant d'une démarche plus générale qui consiste à « classifier » les concepts manipulés.

### 1.1.1 Classer les objets

La généralisation est un mécanisme qui consiste à réunir des objets possédant des caractéristiques communes dans une nouvelle classe plus générale appelée `super-classe`.

Prenons un exemple décrit par la figure **??**. Nous disposons dans un diagramme de classes initial d'une classe `Voiture` et d'une classe `Bateau`. Une analyse de ces classes montre que leurs objets partagent des attributs et des opérations : on abstrait une super-classe `Véhicule` qui regroupe les éléments communs aux deux sous-classes `Voiture` et `Bateau`. Les deux sous-classes héritent les caractéristiques communes définies dans leur super-classe `Véhicule` et elles déclarent en plus les caractéristiques qui les distinguent (ou bien elles redéfinissent selon leur propre point de vue une ou plusieurs caractéristiques communes).

Le mécanisme dual, la spécialisation est décrit à partir de l'exemple de la figure **??**.

Ici la spécialisation consiste à différencier parmi les bateaux (la distinction s'effectuant selon leur type), les sous-classes `Bateau_à_moteur` et `Bateau_à_voile`. La spécialisation peut faire apparaître de nouvelles caractéristiques dans les sous-classes.

Il faut retenir que du point de vue de la modélisation, une classe $C_{mere}$ généralise une autre classe $C_{fille}$ si l'ensemble des objets de $C_{fille}$ est inclus dans l'ensemble des objets de $C_{mere}$.

Du point de vue des objets (instances des classes), toute instance d'une sous-classe peut jouer le rôle (peut remplacer) d'une instance d'une des super-classes de sa hiérarchie de spécialisation-généralisation.

### 1.1.2 Discriminants et contraintes

Les relations de spécialisation/généralisation peuvent être décrites avec plus de précision par deux sortes de description UML : les contraintes et les discriminants.

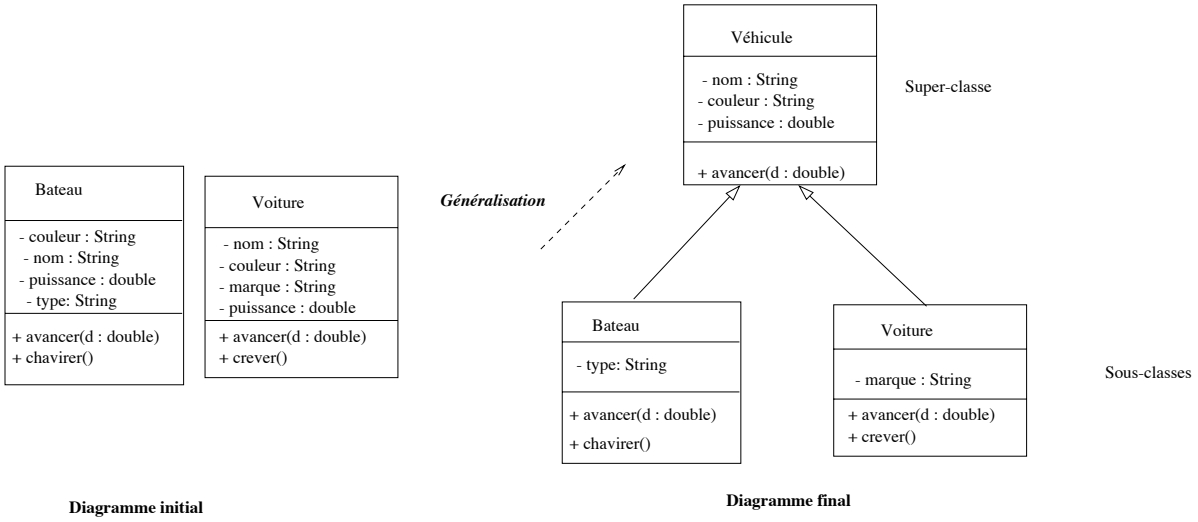


FIGURE 1.1 – Une généralisation

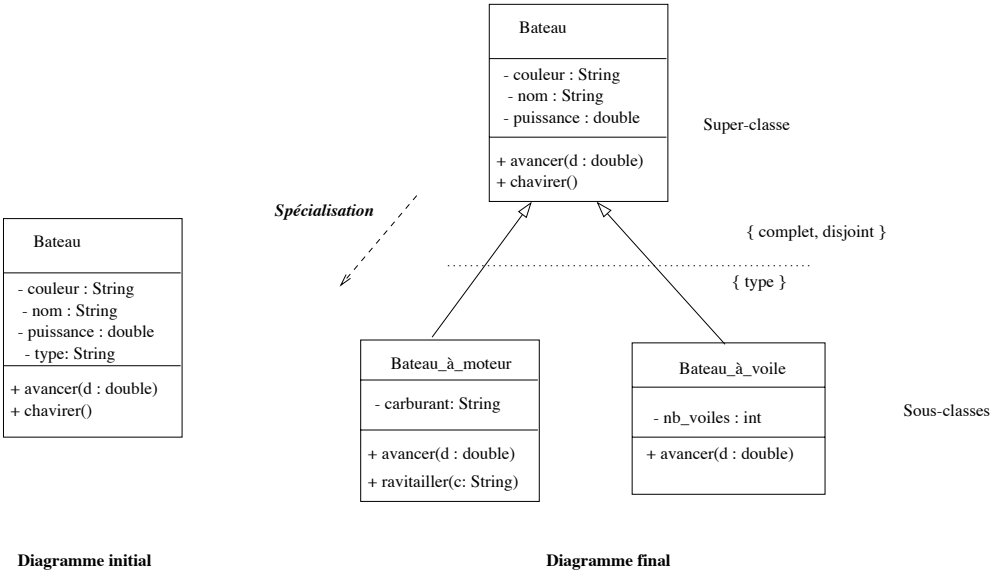


FIGURE 1.2 – Une spécialisation

**Discriminant** Les discriminants sont tout simplement les critères utilisés pour classer les objets dans des sous-classes. Ils étiquettent ainsi les relations de spécialisation et doivent correspondre à une classe du modèle. Dans l'exemple de la figure **??**, deux critères différents de classification sont utilisés : `TypeContrat` partage les employés en salariés et vacataires, tandis que `RetraiteComplémentaire` partage les employés en cotisants et non cotisants. Notez qu'UML autorise qu'un objet soit classé à la fois comme cotisant et comme vacataire : on parle alors de multi-instanciation.

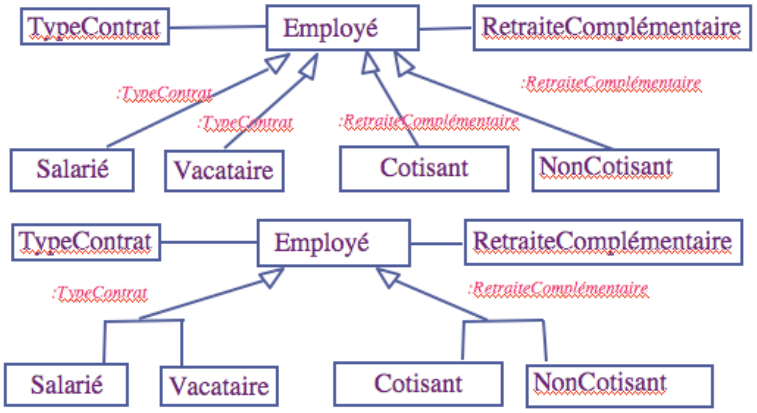


FIGURE 1.3 – Discriminants

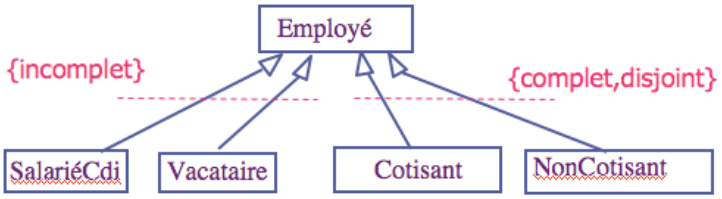


FIGURE 1.4 – Contraintes

**Contraintes** Les contraintes décrivent la relation entre un ensemble de sous-classes et leur super-classe en considérant le point de vue des extensions (ensemble d'instances des classes).

Il en existe quatre, dont trois sont illustrées figure **??** et une figure **??** :

- `incomplete` (`incomplet`) : l'union des extensions des sous-classes est strictement incluse dans l'extension de la super-classe ; par exemple il existe des employés qui ne sont ni salariés, ni vacataires ;
- `complete` (`complet`) : l'union des extensions des sous-classes est égale à l'extension de la super-classe ; par exemple tout employé est cotisant ou non cotisant ;
- `disjoint` : les extensions des sous-classes sont d'intersection vide ; par exemple aucun employé n'est cotisant et non cotisant ;
- `overlapping` (`chevauchement`) : les extensions des sous-classes se rencontrent, par exemple si on avait spécialisé une classe `Vehicule` en `VehiculeTerrestre` et `VehiculeAquatique`, certains véhicules se trouveraient dans les extensions des deux sous-classes.

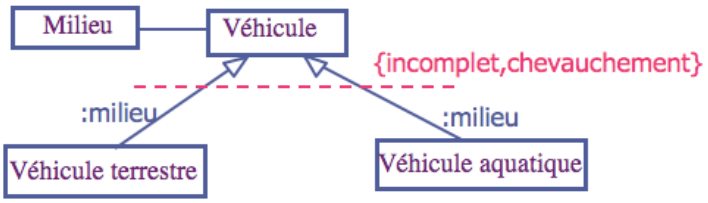


FIGURE 1.5 – Contraintes

## 1.2 Hiérarchie des classes et héritage dans Java

Pour les langages à objets, le programmeur définit des hiérarchies de classes provenant d'une conception dans laquelle il a utilisé le principe de généralisation-spécialisation. On dit le plus souvent que la sous-classe hérite de la super-classe, ou qu'elle étend la super-classe ou encore qu'elle dérive de la super-classe.

Un des avantages des langages à objets est que le code est réutilisable. Il est commode de construire de gros projets informatiques en étendant des classes déjà testées. Le code produit devrait être plus lisible et plus robuste car on peut contrôler plus facilement son évolution, au fur et à mesure de l'avancement du projet. En effet, grâce à la factorisation introduite par la spécialisation-généralisation, on peut modifier une ou plusieurs classes sans avoir à les réécrire complètement (par exemple en modifiant le code de leur super-classe).

Du point de vue des objets (instances de classes), pour Java, une instance d'une sous-classe possède la partie structurelle définie dans la superclasse de sa classe génitrice plus la partie structurelle définie dans celle-ci. Au niveau du comportement, les objets d'une sous-classe héritent du comportement défini dans la superclasse (ou la hiérarchie de super-classes) avec quelques possibilité de variations, comme nous le verrons plus tard. Au niveau de l'exécution, les langages à objets utilisent un mécanisme d'héritage (ou résolution de messages) qui consiste à résoudre dynamiquement l'envoi de message sur les objets (instances), c'est-à-dire à trouver et exécuter le code le plus spécifique correspondant au message.

En Java,

- toutes les classes dérivent de la classe `Object`, qui est la racine de toute hiérarchie de classes.
- une classe ne peut avoir qu'une seule super-classe. On parle d'*héritage simple*. Il existe des langages à objets, tels que C++ ou Eiffel qui autorisent l'héritage multiple.
- le mot clef permettant de définir la généralisation-spécialisation entre les classes est `extends`.

Pour la suite du chapitre nous prenons une hiérarchie de classes représentée dans le diagramme **??**.

Nous commençons par définir la structure de la hiérarchie : vous voyez comment les entêtes des classes incluent le mot clef `extends` pour traduire en Java la relation d'héritage.

```
package Prog.MesExemples;

public class Personne{
......}// fin de la classe Personne
```

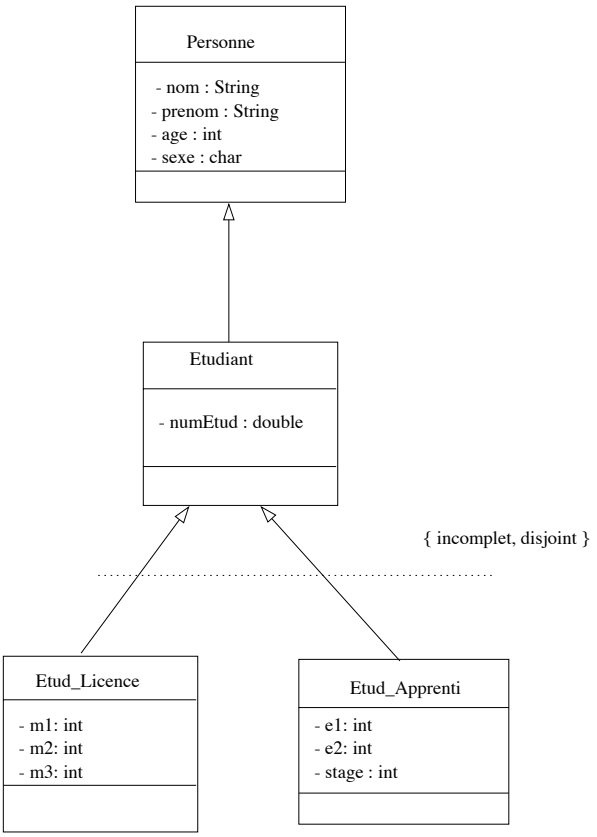


FIGURE 1.6 – Une hiérarchie de classes.

```
.....
public class Etudiant extends Personne {


      ..................


}    // fin de la classe Etudiant

.............
public class Etud_Licence extends Etudiant {


      ..................


}    // fin de la classe Etud_Licence
.............
public class Etud_Apprenti extends Etudiant {


      ..................
```

```
}    // fin de la classe Etud_Apprenti
```

Tous les étudiants sont des personnes, mais on peut les spécialiser en étudiants de licence ou étudiants apprentis. Les attributs définis dans la classe `Personne` sont nom, prénom, age, sexe. Bien sûr, on définit pour cette classe les constructeurs, les accesseurs, et une opération ou méthode `incrementerAge()`(cette méthode vieillit d'un an).

La classe `Etudiant` possède l'attribut supplémentaire `numEtud` qui représente le numéro de l'étudiant. La classe `Etudiant` définit des opérations de calculs de moyenne, d'admission, de mention.

Les étudiants de licence (classe `Etud_Licence`) suivent 3 modules (M1, M2, M3) de coefficient 1. Ils sont admis si leur moyenne générale est supérieure ou égale à 10 et ils conservent les modules obtenus s'ils ne sont pas admis.

Les étudiants apprentis (classe `Etud_Apprenti`) suivent 2 modules (E1, E2) de coefficient 1 et un stage de coefficient 2. Ils sont admis si leur moyenne est supérieure ou égale à 10 et si leur note de stage est supérieure à 8.

Une instance étudiant apprenti (ou une instance d'étudiant licence) a la possibilité d'utiliser des méthodes définies dans Personne :

```
//exemple de code pouvant figurer dans la méthode main d'une classe application
Etud_Apprenti a = new Etud_Apprenti ("Einstein", "Albert", 22, 'M', 123, 8, 8, 10);
// L'age de cet apprenti est 22 ans
a.incrementerAge();
System.out.println(a.getName()+"était âgé de " + a.getAge()+" ans");
```

A l'exécution, on obtient :

```
Einstein Albert était âgé de 23 ans
```

## 1.3 Redéfinition de méthodes - Surcharge - Masquage

Une opération (ou une méthode) est déclarée dans une classe, possède un nom, un type de retour et une liste de paramètres, on parle de signature pour désigner l'ensemble des informations (nom de classe + nom de l'opération + type de retour + liste des paramètres). A priori dans une hiérarchie de classes, chaque classe ayant un nom différent, deux opérations de même signature sont interdites. Mais le programmeur peut :

- au sein d'une même classe donner le même nom à une opération si la liste des paramètres est différente,
- dans des classes distinctes, donner le même nom et la même liste de paramètres. Lorsqu'une méthode d'une sous-classe a même nom et même liste de paramètres qu'une méthode d'une super-classe, on dit que la méthode de la sous-classe *redéfinit* la méthode de la super-classe.

On parle alors de *surcharge*. Lorsque le long d'un chemin de spécialisation-généralisation on rencontre des opérations de même nom et même liste de paramètres on parle de *masquage*. Donc lorsque l'on code une sous-classe, on a la possibilité de masquer la définition d'une (ou de plusieurs) méthode(s) de la super-classe, simplement en redéfinissant une méthode qui a le même nom et les mêmes paramètres que celle de la super-classe.

*Remarque :* La méthode de la super-classe sera encore accessible mais en la préfixant par le mot-clé `super`, pseudo-variable (car définie par le système).

```
// Classe Etudiant
.............
public class Etudiant extends Personne{
.............
public boolean admis()
       {return (moyenne() >= 10;)}
}

// Classe Etud_Apprenti

public class Etud_Apprenti extends Etudiant {

.............
public boolean admis()
       {
        return (super.admis() && getnoteSt() > 8);
        // on fait appel ici à l'opération admis() définie dans Etudiant
        //  on aurait pu aussi définir admis dans Etud_Apprenti par
        // return (moyenne() >= 10 && getnoteSt() > 8;)
        }
}
```

*Remarque :* La forme de l'opération `admis` qui utilise `super` est plus stable et plus réutilisable, en effet si les conditions d'admission générale sur tous les étudiants changent, le code de l'opération `admis` dans la classe `Etudiant` sera modifié, entraînant la modification automatique de toutes les opérations qui font appel à elle via super.admis().

## 1.4 Les constructeurs

Dès qu'un constructeur a été défini dans une classe de la hiérarchie de classes, il est nécessaire de définir des constructeurs lui correspondant dans toutes les sous-classes. D'autre part, il est commode et intéressant de faire appel aux constructeurs des super-classes que l'on invoque par le mot clé `super` suivi de ses arguments entre parenthèses. L'invocation du constructeur de la super-classe doit se faire obligatoirement à la première ligne.

```
// la classe Personne

public Class Personne{
.............
// constructeur par défaut
public Personne() {
nom = "";
prenom = "";
age = 0;
sexe = 'F';}
```

```
// autre constructeur
public Personne(String n, String p, int a, char s) {
nom = n;
prenom = p;
age = a;
sexe = s;}
// la classe Etudiant
public Class Etudiant extends Personne{
.............
// constructeur par défaut
public Etudiant()
       {
       // super() sera automatiquement réalisé
        numEtu=-1;
        }

// autre constructeur
public Etudiant(String n, String p, int a, char s, double n)
        {
        super(n, p, a, s); // appel au 2ième constructeur
                                    // de la super-classe Personne;
        numEtu=n;
        }


// la classe Etu_Apprenti
public Class Etu_Apprenti extends Etudiant{
// constructeur par défaut
public Etu_Apprenti ()
       {
       // Par défaut, super() est appelée.
       E1=-1; E2=-1; stage=-1;
       }
// autre constructeur
public Etu_Apprenti (String n, String p, int a, char s, double n, int e1, int e2, int s
        {
        super(n, p, a, s, n); // appel au 2ième constructeur de Etudiant
        E1 = e1; E2 = e2; stage = st;
      }
```

*Remarque :* L'utilisation de `super` dans les constructeurs permet de mieux gérer l'évolution du code, une modification d'un constructeur d'une super-classe provoquant automatiquement la modification des constructeurs des sous-classes faisant appel à lui par `super`.

## 1.5 Protections

Nous avons déjà introduit les directives de protection, nous pouvons donc compléter ici leur description (quoique le mécanisme d'accès reste encore bien obscur) :

— `public` : la méthode (ou l'attribut) est accessible par tous et de n'importe où.

— `protected` : la méthode (ou l'attribut) n'est accessible que par les classes du même package et par les sous-classes (même si elles se trouvent dans un autre package). *Remarque :* pour le contrôle d'accès `protected`, il semblait logique de n'autoriser l'accès qu'à la classe concernée et à ses sous classes (comme en C++).
— `private` : la méthode (ou l'attribut) est accessible uniquement par les méthodes de la classe. *Remarque :* Cependant les instances d'une même classe ont accès aux méthodes et attributs privés des autres instances de cette classe.

Lorsque rien n'est précisé, l'accès est autorisé depuis toutes les classes du même package.

## 1.6 Classes et méthodes abstraites

Dans une hiérarchie de classes, plus une classe occupe un rang élevé, plus elle est générale donc plus elle est abstraite. On peut donc envisager de l'abstraire complétement en lui ôtant d'une part le rôle de génitrice (elle ne sera pas autorisée à créer des instances) et en lui permettant d'autre part de factoriser structures et comportements (sans savoir exactement comment les faire) uniquement pour rendre service à sa sous-hiérarchie.

Une méthode *abstraite* est une méthode déclarée avec le mot clef `abstract` et ne possède pas de corps (pas de définition de code). Par exemple, le calcul de la moyenne d'un étudiant ne peut pas être décrit au niveau de la classe `Etudiant`.

Néanmoins, on sait que les étudiants de licence et les étudiants apprentis doivent être capable de calculer leur moyenne. La méthode `moyenne` doit être déclarée abstraite au niveau de la classe `Etudiant`. Par contre, il faudra obligatoirement définir le corps de la méthode `moyenne` dans les sous-classes de la classe `Etudiant`.

Si une classe contient au moins une méthode abstraite, alors il faut déclarer cette classe abstraite, mais on peut aussi définir des classes abstraites (parce qu'on ne veut pas qu'elles engendrent des objets) sans aucune méthode abstraite.

```
.........
// la classe est déclarée abstraite
public abstract class Etudiant extends Personne{

.........
// la méthode est déclarée abstraite
public abstract double moyenne();
}
```

*Remarques :* Dans le formalisme UML les classes et méthodes abstraites ont leur nom en italique.

Une classe abstraite peut être sous-classe d'une classe concrète (ici `Etudiant` est une sous-classe abstraite de `Personne` qui est concrète).

L'intérêt de définir une méthode abstraite est double : il permet au développeur de ne pas oublier de redéfinir une méthode qui a été définie `abstract` au niveau d'une des super-classes ; il permet de mettre en œuvre le polymorphisme.

## 1.7 Le polymorphisme

### 1.7.1 Transformation de type ou Casting

Une référence sur un objet d'une sous-classe peut toujours être implicitement convertie en une référence sur un objet de la super-classe.

```
Personne p;
Etud_Licence e=new Etud_Licence();
p=e;
// tout étudiant de licence est un étudiant et a fortiori une personne
```

L'opération inverse (cast-down) est possible de manière explicite (mais avec précaution et uniquement en cas de nécessité absolue).

```
Etud_Licence e;
Personne p=new Etud_Licence();
e = (Etud_Licence) p ;
// p peut désigner des personnes (type statique ou type de la variable)
// p désigne ici en realite un etudiant de licence
// (type dynamique ou type défini par le new)
```

Cette opération de transformation de type explicite peut être utilisée indépendamment des hiérarchies de classes sur les types primitifs.

```
double x = 9.9743;
int xEntier = (int) x; // alors xEntier=9
```

Pour vérifier qu'une instance appartient bien à une classe précise d'une hiérarchie, on peut utiliser l'opérateur `instanceof`. Par exemple,si on veut tester si l'instance `e` a pour classe ou pour superclasse la classe `Etud_Apprenti`,on écrit :

```
 if (e instanceof Etud_Apprenti) .....
```

### 1.7.2 Polymorphisme des opérations

Le fait que l'on puisse définir dans plusieurs classes des méthodes de même nom revient donc à désigner plusieurs formes de traitement derrière ces méthodes. Le code de la méthode qui sera réellement exécuté n'est donc pas figé, un appel de message autorisé à la compilation donnera des résultats différents à l'exécution car le langage retrouvera selon l'objet et la classe à laquelle il doit son existence le code à exécuter (on parle de `liaison dynamique`). La capacité pour un message de correspondre à plusieurs formes de traitements est appelé `polymorphisme`

Quelques exemples pour fixer les idées.

```
\\une autre hiérarchie de classes

public abstract class Felin {
......
       public abstract String pousseSonCri();
......
 }

public class Lion extends Felin {
......
       public String pousseSonCri() {return "rouaaaaah";}
```

```
        public String toString() {return "lion";}
......
}

public class Chat extends Felin {
......
        public String pousseSonCri() {return "miaou");}
        public String toString() {return "chat";}
......
}

public class AppliCriDeLaBete {
        public static void main(String args[]) {
        Felin tab[] = new Felin[2];
        tab[0] = new Lion();
        tab[1] = new Chat();
        for( int i=0; i<2; i++)
          System.out.println("Le cri du "+tab[i]+" est : "+ tab[i].pousseSonCri();
        }
}
```

Remarquer ici, que le main utilise un tableau de Felin, que le casting implicite est utilisé, et que tab[i] étant un Felin pour le compilateur et la méthode pousseSonCri étant définie dans Felin (sous forme abstraite), il n'y a pas d'erreur de compilation et à l'exécution, on obtient :

```
Le cri du lion est : rouaaaaah
Le cri du chat est : miaou
```

Nous montrons à présent un exemple de polymorphisme dans le cas d'une classe représentant les promotions d'étudiants.

```
..........
public class Promotion {
private ArrayList<Etudiant> listeEtudiants = new ArrayList<Etudiant>();
.....
public double moyenneGenerale() {
        double somme = 0;
        for (int i=0;i<listeEtudiants.size();i++)
                {Etudiant etud = listeEtudiants.get(i));
// Le polymorphisme a lieu ici: le calcul de la méthode moyenne() sera
// différent si etud est une instance de Etud_Licence ou de Etud_Apprenti.
                somme = somme + etud.moyenne();
                }
            }
        if (listeEtudiants.size()>0)
                        return (somme / listeEtudiants.size());
        else return 0;
   }
```

```
public String nomDesEtudiants() {
       String listeNom = "";
// Le polymorphisme a lieu ici: la méthode toString() est appelée sur tous les
// elements  de la ArrayList listeEtudiants.

       for (int i=0;i<listeEtudiants.size();i++)
                listeNom = listeNom +listeEtudiants.get(i).toString() + " \n";

       return listeNom;
   }
 } // fin classe Promotion



public class AppliPromo {
    public static void main(String args[]) {
    Promotion promo = new Promotion (2000);
    Etud_Licence e1 = new Etud_Licence("Cesar", "Julio", 26, 'M', 127, 14, 12, 10);
    promo.inscrit (e1);
    Etud_Licence e2 = new Etud_Licence("Curie", "Marie", 22, 'F', 124, 8, 17, 20);
    promo.inscrit (e2);
    Etud_Apprenti a1 = new Etud_Apprenti ("Bol", "Gemoï", 22, 'M', 624, 8, 8, 10);
    promo.inscrit (a1);
    Etud_apprenti a2 = new Etud_Apprenti ("Einstein", "Albert", 22, 'M', 123, 13, 17, 1
    promo.inscrit (a2);

    System.out.println("La moyenne de la promotion est : " + promo.moyenneGenerale() );
    promo.incrementeNoteE1DesApprentis();
    System.out.println("La nouvelle moyenne de la promotion est : "
                                          + promo.moyenneGenerale() );
    System.out.println("Les étudiants de la promotion sont : "
                                          + promo.nomDesEtudiants() );
       }
}
```

A l'exécution, on obtient, en supposant que les méthodes `toString` des classes représentant les étudiants insèrent le type des étudiants (licence ou apprenti) :

```
La moyenne de la promotion est : 13
La nouvelle moyenne de la promotion est : 13.5
Les étudiants de la promotion sont :
Cesar Julio (Etud_Licence)
Curie Marie (Etud_Licence)
Bol Gemoï (Etud_Apprenti)
Einstein Albert (Etud_Apprenti)
```

En général, il est préférable de ne pas avoir à recourir à la transformation de type explicite. Dans notre exemple Etudiant, il n'est pas nécessaire de caster les étudiants d'une liste d'étudiants en Etud_Apprenti ou Etud_Licence pour calculer leur moyenne().

Pour faire court, le polymorphisme fonctionne grâce à la liaison dynamique qui cherche à l'exécution la méthode la plus spécifique pour résoudre un envoi de message. Cette méthode plus spécifique se trouve soit dans la classe de l'instance (classe déterminée par le type dynamique suivant le `new`), soit dans une super-classe, en remontant à partir de la classe de l'instance et en utilisant la première méthode trouvée répondant au message.