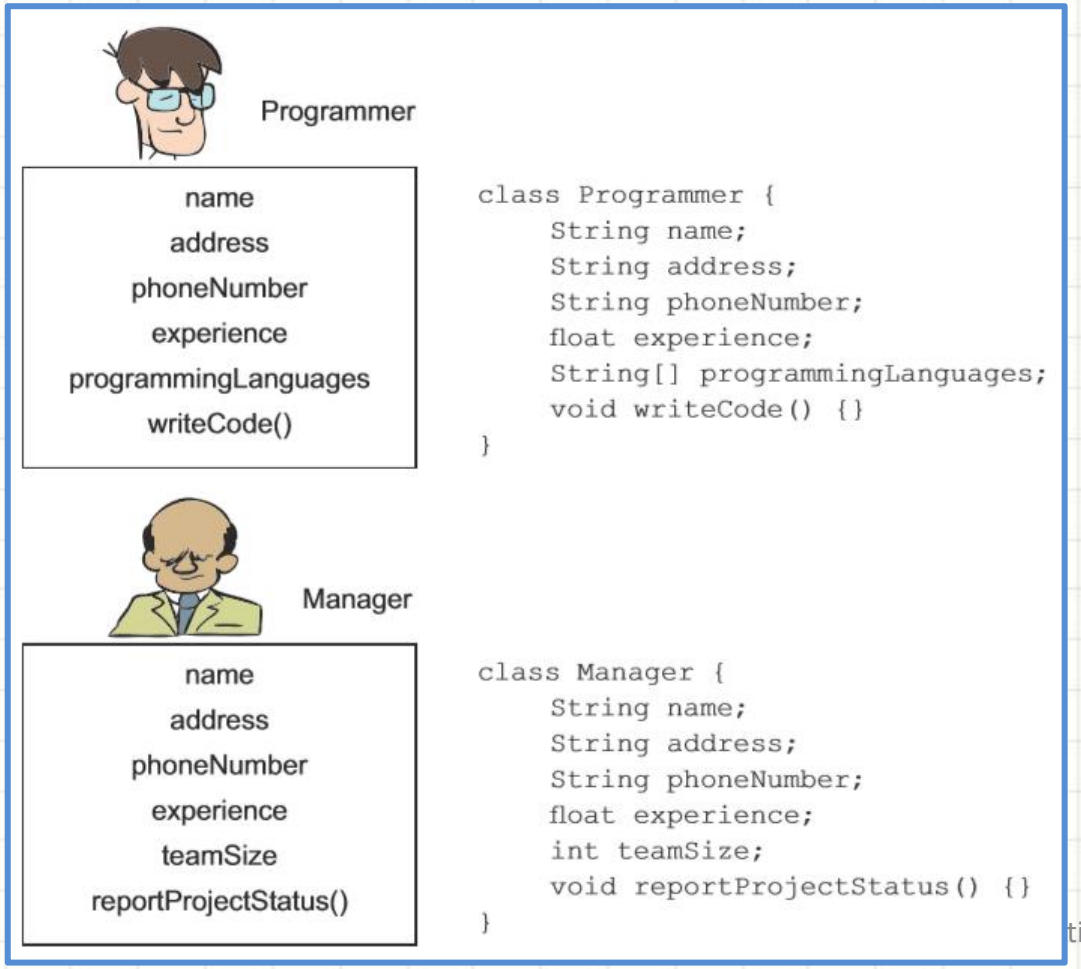
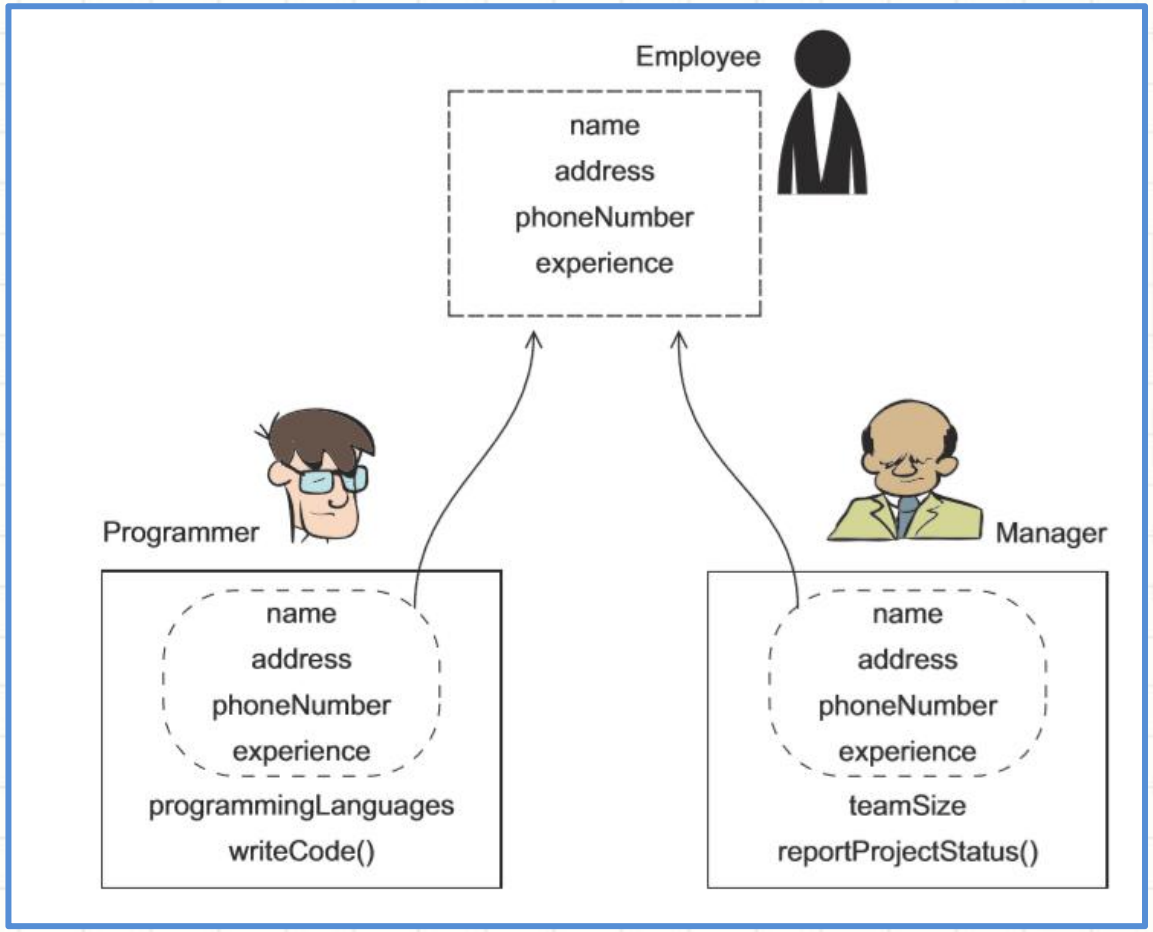
PROGRAMMATION ORIENTÉE OBJET CHAP5: HÉRITAGE ET POLYMORPHISME

L'héritage: Pourquoi?

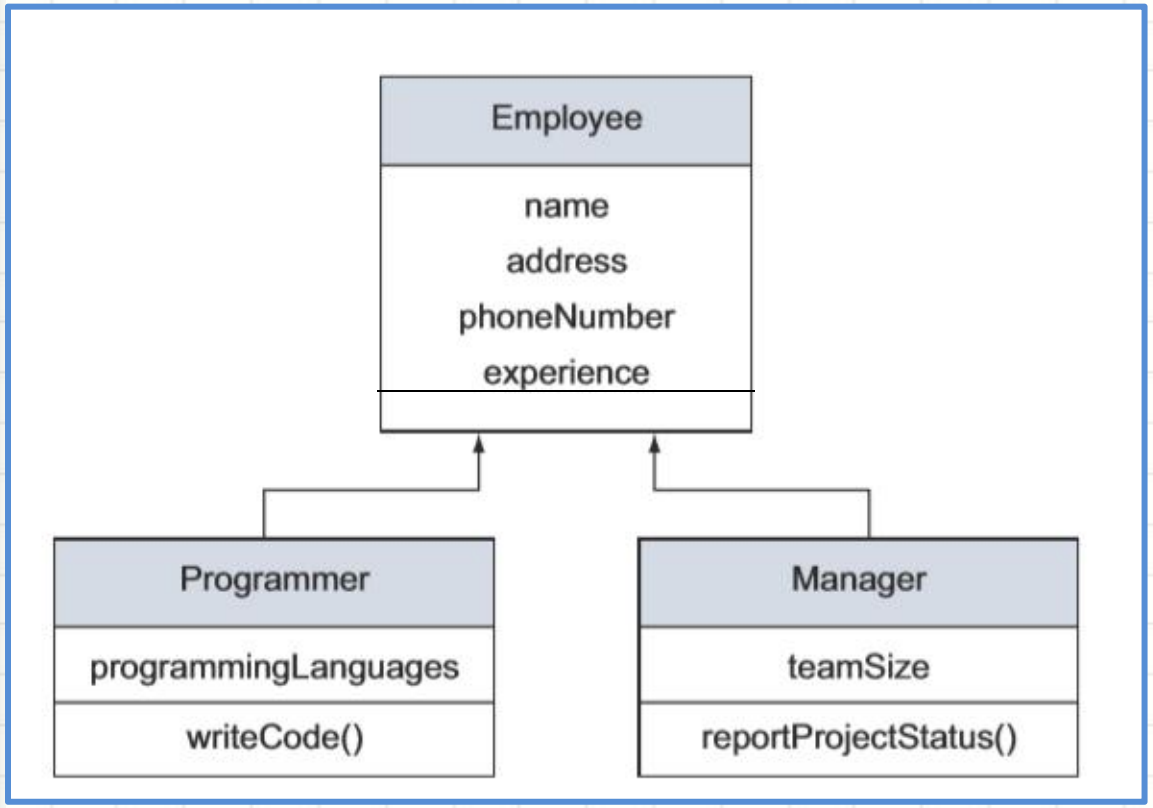
- Considérons cet exemple:





Source: MALA Gupta. OCA JAVA SE 7. Programmer I Certification Guide

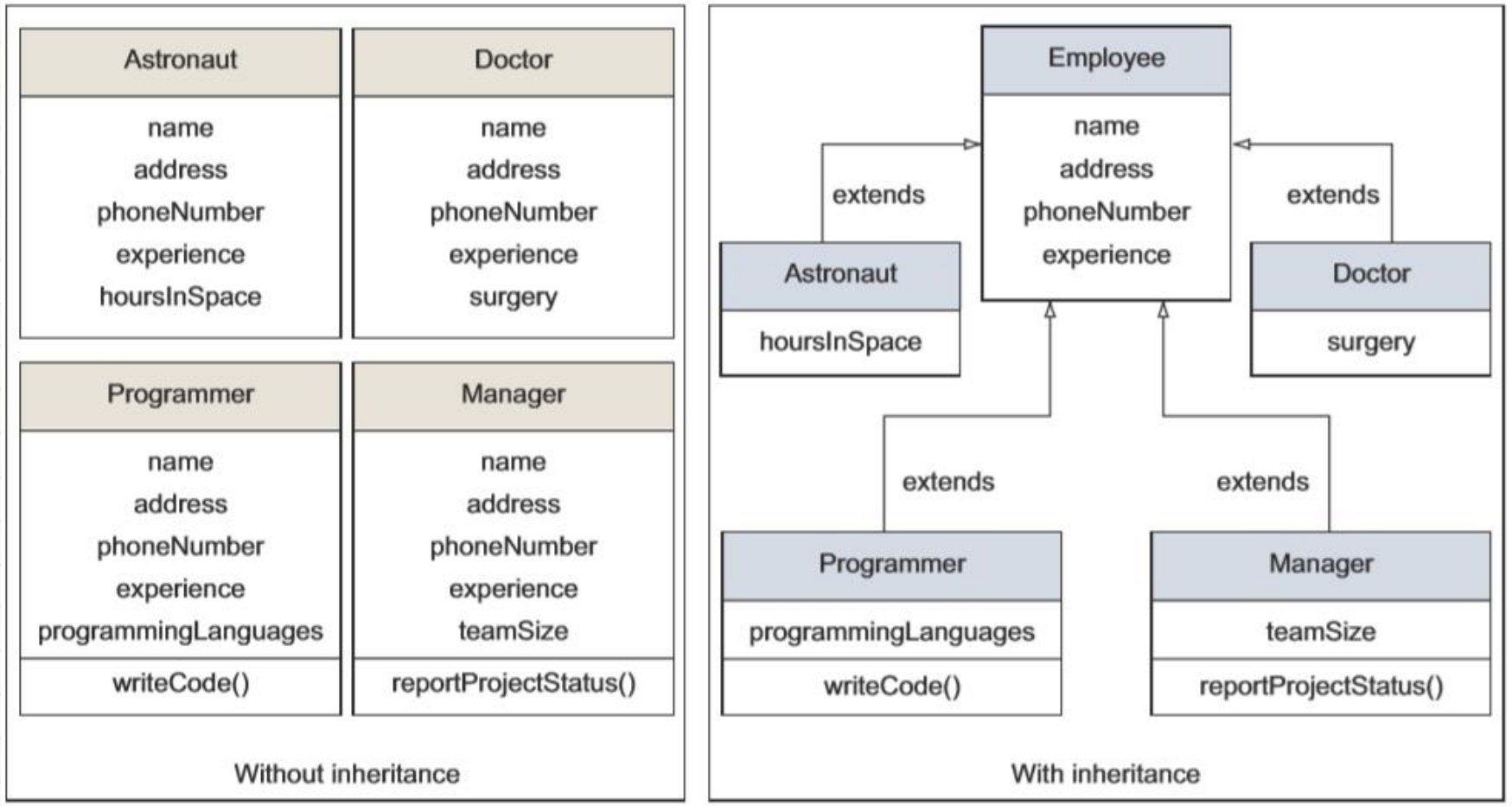
L’héritage: Pourquoi?




```
class Employee {
     String name;
     String address;
     String phoneNumber;
     float experience;
}
class Programmer extends Employee {
     String[] programmingLanguages;
     void writeCode() {}
}
class Manager extends Employee {
     int teamSize;
     void reportProjectStatus() {}
}
```

Source: MALA Gupta. OCA JAVA SE 7. Programmer I Certification Guide

L’héritage: Pourquoi?



Source: MALA Gupta. OCA JAVA SE 7. Programmer I Certification Guide

L’héritage: Introduction

- Le concept d’héritage constitue l’un des fondements de la P.O.O. En particulier, il est à la base des possibilités de réutilisation de composants logiciels (en l’occurrence, de classes).
- En effet, il vous autorise à définir une nouvelle classe, dite **« dérivée »**, à partir d’une classe existante dite **« de base »**.
- L’héritage permet de **hiérarchiser** les classes et les objets. Il est basé sur l’idée qu’un **objet spécialisé** bénéficie ou **hérite** des caractéristiques de **l’objet le plus général** auquel il rajoute ses éléments propres.

L’héritage: Vocabulaire

- En terme de concepts objets cela se traduit de la manière suivante:
  - Nous associons une classe au concept le plus général que nous l’appellerons **classe de base** ou **classe mère** ou **super-classe.**
  - Pour chaque concept spécialisé, on **dérive une classe du concept de base.**
  - La nouvelle classe est dite **classe dérivée** ou **classe fille** ou **sous-classe**
  - L’héritage dénote une relation de **généralisation / spécialisation**, on peut traduire une relation d’héritage par la phrase :

« La classe dérivée est une version spécialisée de sa classe de base »

L'héritage: comment l'identifier?

- On est sûr qu'il y a héritage entre une classe A et une classe B lorsqu'on peut dire que:

  « A est une version spécialisée de B »

  ou encore **« A est un B »** **« A is-A B »**

- **Exemples:**
  - Une voiture est un véhicule (Voiture hérite de Véhicule)
  - Un bus est un véhicule (Bus hérite de Véhicule)
  - Un chat est un mammifère (Chat hérite de Mammifère)
  - Un corbeau est un oiseau (Corbeau hérite de Oiseau)
  - Un perroquet est un oiseau (Perroquet hérite de Oiseau)
  - Un chirurgien est un médecin (Chirurgien hérite de Médecin)

L'héritage: Accès aux membres de la classe de base

```
class Point
{
  public void initialise (int abs, int ord)
  { x = abs ; y = ord ;
  }

  public void deplace (int dx, int dy)
  { x += dx ; y += dy ;
  }

  public void affiche ()
  { System.out.println ("Je suis en " + x + " " + y) ;
  }
  private int x, y ;
}
```

```
class Pointcol extends Point     // Pointcol dérive de Point
{ public void colore (byte couleur)
  { this.couleur = couleur ;
  }
  private byte couleur ;
}
```

Source: Claude Delannoy. Programmer en Java. 6ème édition. Eyrolles. 2008

```
public class TstPcol1
{ public static void main (String args[])
  { Pointcol pc = new Pointcol() ;
    pc.affiche() ;
    pc.initialise (3, 5) ;
    pc.colore ((byte)3) ;
    pc.affiche() ;

    pc.deplace (2, -1) ;
    pc.affiche() ;
    Point p = new Point() ; p.initialise (6, 9) ;
    p.affiche() ;
  }
}
```

```
Je suis en 0 0
Je suis en 3 5
Je suis en 5 4
Je suis en 6 9
```

L'héritage: Accès aux membres de la classe de base

- **Un objet d'une classe dérivée accède aux membres publics de sa classe de base**, exactement comme s'ils étaient définis dans la classe dérivée elle-même.
- Un objet de type Pointcol peut alors faire appel :
  - aux méthodes publiques de Pointcol, ici colore() ;
  - mais aussi aux méthodes publiques de Point : initialise(), deplace() et affiche().
- **Une méthode d'une classe dérivée n'a pas accès aux membres privés de sa classe de base.**
- **Une méthode d'une classe dérivée a accès aux membres protégés de sa classe de base.**

L’héritage: Accès aux membres de la classe de base

- **Protected (#)**: est un droit d’accès qu’on peut classer entre public (le plus permissif) et private (le plus restrictif). Il n’a de sens que pour les classes mères qui se font hériter mais on peut l’utiliser même quand il n’y a pas héritage.
- Sa signification est la suivante: les éléments protégés ne seront **accessibles depuis l’extérieur que pour une classe fille ou une classe du même package.**

Construction et initialisation des objets dérivés

```
class Point
{
  public Point(int abs, int ord)
  { x = abs ; y = ord ;
  }

  public void deplace (int dx, int dy)
  { x += dx ; y += dy ;
  }

  public void affiche ()
  { System.out.println ("Je suis en " + x + " " + y) ;
  }
  private int x, y ;
}
```

```
public class TstPcol3
{ public static void main (String args[])
  { Pointcol pc1 = new Pointcol(3, 5, (byte)3) ;
    pc1.affiche() ;   // attention, ici affiche
    pc1.affichec() ;  // et ici affichec

    Pointcol pc2 = new Pointcol(5, 8, (byte)2) ;
    pc2.affichec() ;
    pc2.deplace (1, -3) ;
    pc2.affichec() ;
  }
}
```

```
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
  { super (x, y) ;                // obligatoirement comme première instruction
    this.couleur = couleur ;
  }
  public void affichec ()
  { affiche() ;
    System.out.println ("  et ma couleur est : " + couleur) ;
  }
  private byte couleur ;
}
```

```
Je suis en 3 5
Je suis en 3 5
  et ma couleur est : 3
Je suis en 5 8
  et ma couleur est : 2
Je suis en 6 5
  et ma couleur est : 2
```


Source: Claude Delannoy. Programmer en Java. Eyrolles. 2008

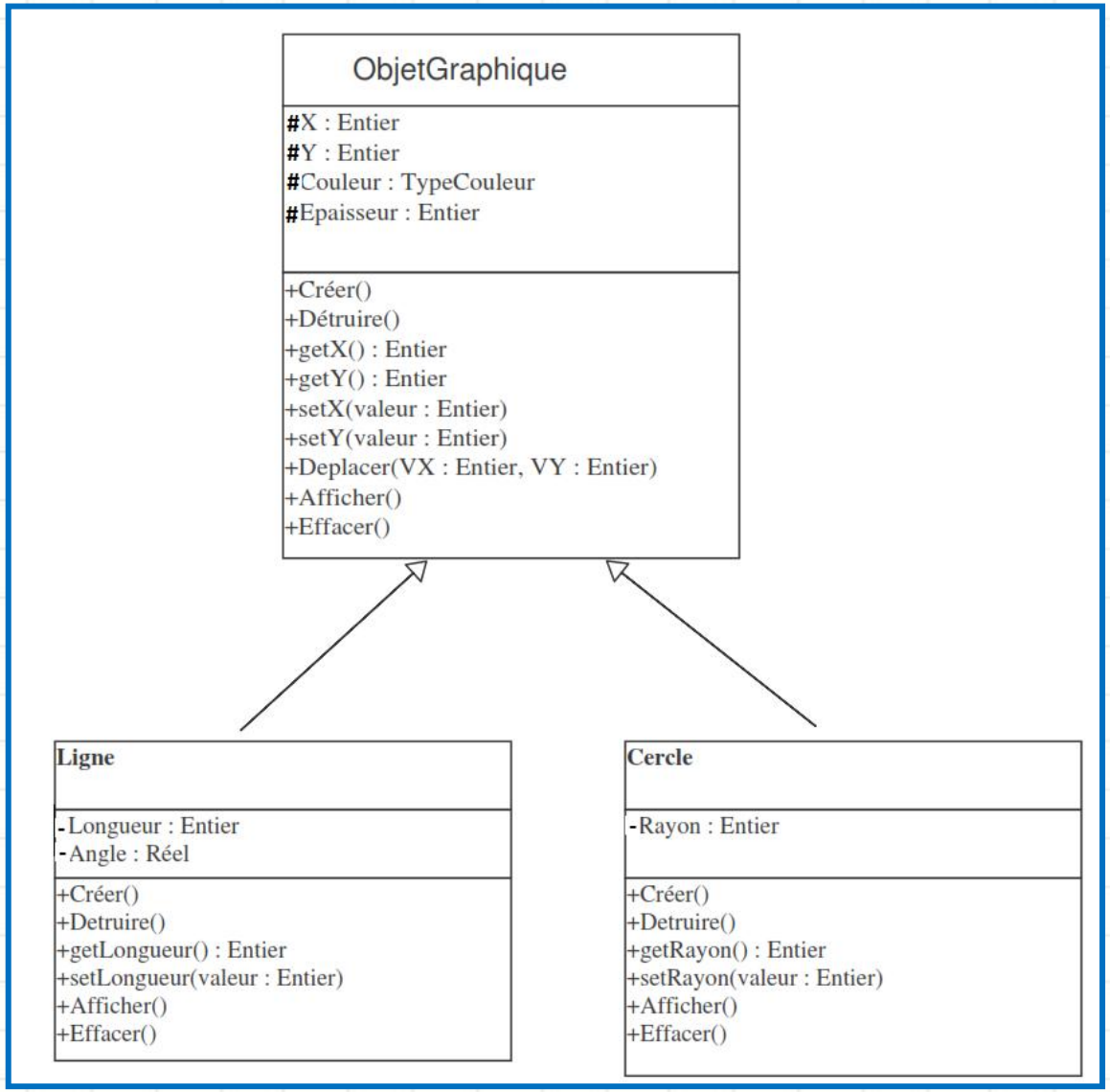
Construction et initialisation des objets dérivés

- **Un constructeur de la classe mère n'est pas hérité de la classe dérivée.**
- En Java, le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet.
- Si un constructeur d'une classe dérivée appelle un constructeur d'une classe de base, il **doit obligatoirement s'agir de la première instruction du constructeur et ce dernier est désigné par le mot clé super**.

L’héritage: exemple

- **Exemple : les objets graphiques**
- Considérons un ensemble d’**objets graphiques**. Chaque objet graphique peut être considéré relativement à un point de base que nous représenterons par ses coordonnées cartésiennes X et Y.
- On lui associe également sa couleur ainsi que l’épaisseur du trait.
- Hormis la création d’objets, nous associons les méthodes suivantes à notre objet graphique :
  - accès en lecture et en écriture des attributs
  - affichage
  - effacement
  - déplacement d’un objet.
- Rajoutons deux classes spécialisées : **Ligne** et **Cercle**.
- Chacune d’entre elles rajoute ses attributs propres : le rayon pour le cercle, la longueur et l’angle pour la ligne.

L’héritage simple: exemple (UML)



L’héritage: exemple

- Ainsi, les classes Ligne et Cercle disposent de leurs attributs propres qui traduisent leur spécialisation et des attributs de la classe de base qui sont hérités.
- Les méthodes de la classe de base sont également héritées. Les classes Ligne et Cercle n’ont pas, par exemple, à fournir de code pour la méthode getX() chargée de renvoyer la valeur de l’attribut X.
- En revanche, elles sont libres de **rajouter** les méthodes qui leur sont propres, par exemple, des méthodes pour accéder aux attributs supplémentaires.
- Une classe dérivée peut **redéfinir** une méthode de la classe mère: c’est à dire fournir une nouvelle définition d’une méthode de la classe ascendante.

Redéfinition

- Nous avons déjà étudié la notion de **surcharge / surdéfinition (overloading)** de méthodes à l'intérieur d'une même classe.
- Nous avons vu qu'elle correspondait à des méthodes de même nom, mais de signatures différentes.
- Dans le cadre de l'héritage, on parle de **redéfinition (overriding)** il s'agira non seulement de méthodes de même nom (comme pour la surcharge), mais aussi de même signature et de même type de valeur de retour.
- Alors que la surcharge permet de cumuler plusieurs méthodes de même nom, la redéfinition substitue une méthode à une autre.

Redéfinition

```
class Pointcol extends Point
{  public Pointcol (int x, int y, byte couleur)
   {  super (x, y) ;                  // obligatoirement comme première instruction
      this.couleur = couleur ;
   }
   public void affiche ()
   {  super.affiche() ;
      System.out.println ("  et ma couleur est : " + couleur) ;
   }
   private byte couleur ;
}
```

```
public class TstPcol4
{  public static void main (String args[])
   {  Pointcol pc = new Pointcol(3, 5, (byte)3) ;
      pc.affiche() ;  // ici, il s'agit de affiche de Pointcol
      pc.deplace (1, -3) ;
      pc.affiche() ;
   }
}
```

```
Je suis en 3 5
  et ma couleur est : 3
Je suis en 4 2
  et ma couleur est : 3
```

Source: Claude Delannoy. Programmer en Java. Eyrolles. 2008

Exercice

I. Déclarer une classe Animal à 2 champs : un entier age et une chaine de caractères nom_du_cri. Veiller à déclarer les champs protected.
   1. Ajouter un constructeur par défaut et un constructeur à 2 paramètres.
   2. Définir une méthode vieillir() qui augmente d'une unité l'âge du chien et le retourne.
   3. Déclarer et définir une méthode presenter() qui affiche l'age et le type de cri de l'animal : aboiement, henissement, roucoulement, … sous cette forme :

      « L'animal a ***age*** ans et son cri s'appelle ***nom_du_cri*** »

Exercice (suite)

II. Définir maintenant une classe fille Chien qui hérite de la classe Animal.

On suppose ici que l'aboiement du chien dégage différentes sonorités : « waaf », « woof » ou « grrrh », …

1. Ajouter à la classe fille un champ cri, un constructeur paramétrique, un constructeur par défaut. Le champ nom_du_cri est initialisé à « aboie » au niveau du constructeur par défaut.
2. Redéfinir dans cette classe fille la méthode presenter(). Elle présentera l'animal différemment suivant son âge :
   - S'il est jeune (moins de 6 ans) : «Le chien a ***age*** ans et aboie : ***cri cri cri*** »
   - Sinon « Le chien a ***age*** et aboie ***cri*** »

Les classes et les méthodes « final »

- Lors de la conception, on peut choisir d'interdire l'héritage d'une classe donnée; celle-ci est alors dite finale.

final public class A {}

- Une classe ***final*** est une classe qui ne peut pas avoir de filles. Une classe final ne peut pas être étendue: le mécanisme d'héritage est bloqué.
- Mais une classe final peut évidemment être la fille d'une autre classe (non final).
  - Exemple: la classe **String** est **final**

- Une méthode ***final*** est une méthode qui ne peut pas être redéfinie dans les sous-classes.

```
class A {
    final public void f() {}
}
class B extends A {
// B ne peut pas redéfinir f() !
}
```

Les classes abstraites

- Une classe abstraite est totalement l'opposé d'une classe final: Une classe abstraite est une classe qui ne peut pas être instanciée directement. Elle n'est utilisable qu'à travers sa descendance.
- Une classe abstraite doit toujours être dérivée pour pouvoir générer des objets.
- Une classe abstraite est précédée du mot clé ***abstract***
- L'utilité d'une classe abstraite est de permettre le regroupement des attributs et méthodes communs à un groupe de classes.

Les classes abstraites

- Un classe contenant au moins une méthode abstraite est appelée une classe abstraite et cela doit être explicitement précisé dans la déclaration avec : abstract class
- Une classe abstraite peut contenir des méthodes concrètes.
- Une classe peut être déclarée abstraite sans contenir de méthode abstraite.
- Une classe abstraite ne peut pas être instanciée, car son comportement n’est pas complètement défini.

Les classes abstraites

- Une sous-classe d'une classe abstraite peut :
  1. implémenter toutes les méthodes abstraites. Elle pourra alors être déclarée comme **concrète** et donc **instanciée**;
  2. ne pas implémenter toutes ces méthodes abstraites. Elle reste alors nécessairement abstraite et ne pourra être instanciée;
  3. ajouter d'autre(s) méthode(s) abstraite(s). Elle reste alors nécessairement abstraite et ne pourra être instanciée.

Les classes abstraites: Exercice

- Considérons la classe abstraite Forme:

```
abstract class Forme {
    public abstract float perimetre(); //methode abstraite
    public abstract float surface(); //methode abstraite

    public double rapport() {
        return surface() / perimetre();
    }
}
```

- Implémenter les classes Cercle et Rectangle qui héritent de la classe Forme et qui surchargent la méthode **toString** de **java.lang.Object** en affichant le rayon pour un objet de type Cercle et la longueur et la largeur pour un objet de type Rectangle.

Les classes abstraites: Exercice (Suite)

- Donner le résultat d'exécution de ce programme:

```
public class EssaiFormes {
   public static void main(String[] arg) {
      Cercle cercle = new Cercle(2);
      Rectangle rectangle = new Rectangle(2, 1);

      System.out.printf("Rapport Surface/Perimetre pour " + cercle + ": %.3f\n", cercle.
      rapport());
      System.out.printf("Rapport Surface/Perimetre " + rectangle + " : %.3f\n",
      rectangle.rapport());
   }
}
```

Les interfaces

- Quand toutes les méthodes d'une classe sont abstraites et qu'il n'y a aucun attribut, on aboutit à la notion d'interface.
- **Une interface** est un prototype de classe. Elle définit la signature des méthodes qui doivent être implémentées dans les classes construites à partir de ce prototype.
- Une interface est une "classe" purement abstraite dont toutes les méthodes sont abstraites et publiques. Les mots-clés ***abstract*** et ***public*** sont optionnels.
- Comme les classes abstraites, les interfaces ne sont pas instanciables :
  - Une interface ne possède pas d'attribut instance.
  - Une interface n'a pas de constructeur.

```
interface Forme {
    float perimetre();
    float surface();
}
```

Les interfaces: Implémentation

- On dit qu'une classe implémente une interface si elle définit l'ensemble des méthodes abstraites de cette interface. On utilise alors, dans l'entête de la classe, le mot-clé ***implements*** suivi du nom de l'interface implémentée.
- La classe doit implémenter toutes les méthodes de l'interface, sinon elle doit être déclarée abstract.
- Une classe (abstraite ou non) peut implémenter plusieurs interfaces. La liste des interfaces implémentées doit alors figurer après le mot-clé ***implements*** placé dans la déclaration de classe, en séparant chaque interface par une virgule.

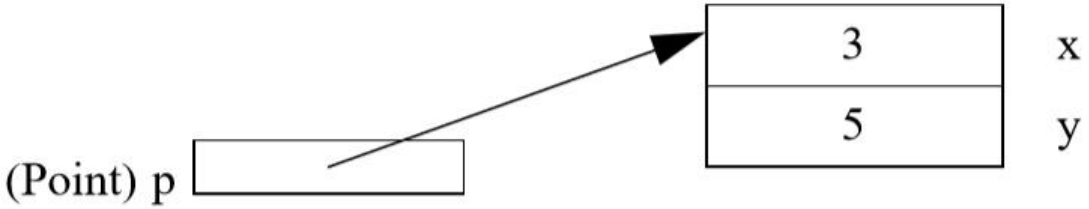
Les interfaces

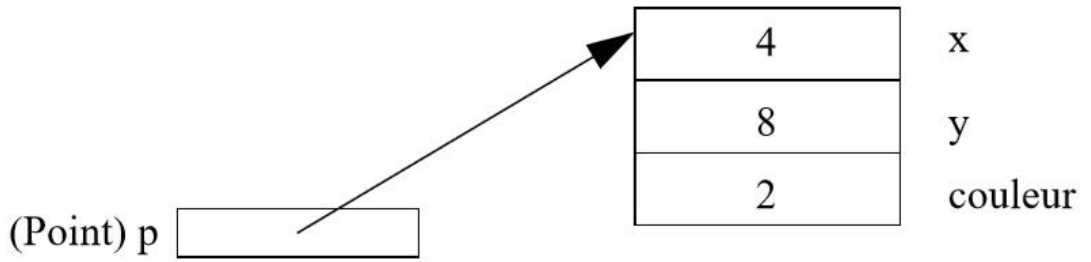
```
class Cercle implements Forme {
   private int rayon;

   public Cercle(int r) {
      this.rayon=r;
   }

   public float perimetre() {
      float resultat = 2 * (float)Math.PI * rayon;
      return resultat;
   }

   public float surface() {
      return (float)Math.PI * rayon * rayon;
   }
}
```

```
class Rectangle implements Forme {
  private int longueur, largeur;

  public Rectangle(int longueur, int largeur) {
    this.longueur = longueur;
    this.largeur = largeur;
  }

  public float perimetre() {
     return 2 * (longueur + largeur);
  }

  public float surface() {
     return longueur * largeur;
  }
}
```

Polymorphisme

Java permet d'affecter à une variable objet non seulement la référence à un objet du type correspondant, mais aussi une référence à un objet d'un type dérivé. Considérons cet exemple:





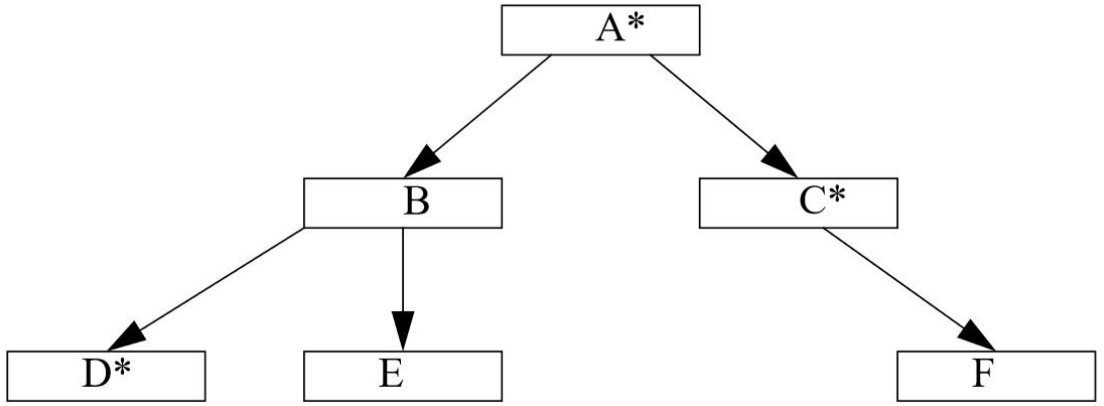

```
Point p = new Point (3, 5) ;
p.affiche () ;      // appelle la méthode affiche de la classe Point
p = new Poincol (4, 8, 2) ;
p.affiche () ;      // appelle la méthode affiche de la classe Pointcol
```

Source: Claude Delannoy. Programmer en Java. 6ème édition. Eyrolles. 2008

Polymorphisme

- La dernière instruction p.affiche() appelle la méthode affiche de la classe Pointcol.
- Autrement dit, elle se base, non pas sur le type de la variable p, mais sur le type effectif de l'objet référencé par p au moment de l'appel (ce type pouvant évoluer au fil de l'exécution).
- Ce choix d'une méthode au moment de l'exécution (et non plus de la compilation) porte généralement le nom de ligature dynamique (ou encore de liaison dynamique).
- En résumé, le polymorphisme en Java se traduit par :
  - la compatibilité par affectation entre un type classe et un type ascendant,
  - la ligature dynamique des méthodes.

Polymorphisme



Avec ces déclarations :

```
A a ; B b ; C c ; D d ; E e ; F f ;


a = b ; a = c ; a = d ; a = e ; a =f ;
b = d ; b = e ;
c = f ;


b = a ;
d = c ;
c = d ;
```

- Les classes marquées d'un astérisque définissent ou redéfinissent une méthode la méthode f().
- Voici quelques exemples précisant la méthode f appelée, selon la nature de l'objet effectivement référencé par a (de type A) :

a référence un objet de type *A* : méthode *f* de *A*

a référence un objet de type *B* : méthode *f* de *A*

a référence un objet de type *C* : méthode *f* de *C*

a référence un objet de type *D* : méthode *f* de *D*

a référence un objet de type *E* : méthode *f* de *A*

a référence un objet de type *F* : méthode *f* de *C*.

Polymorphisme

- Le polymorphisme est ainsi la faculté attribuée à un objet d'être une instance de plusieurs classes. Il a une seule classe "réelle" qui est celle dont le constructeur a été appelé en premier (c'est-à-dire la classe figurant après le new) mais il peut aussi être déclaré avec une classe supérieure à sa classe réelle.
- Cette propriété est très utile pour la creation d'ensembles regroupant des objets de classes différentes comme dans l'exemple suivant :

Polymorphisme

```
interface Forme {
   float perimetre();
   float surface();
}

class Cercle implements Forme {
   private int rayon;

   public Cercle(int r) {
      this.rayon=r;
   }

   public float perimetre() {
      float resultat = 2 * (float)Math.PI * rayon;
      return resultat;
   }

   public float surface() {
      return (float)Math.PI * rayon * rayon;
   }
}
```

```
class Rectangle implements Forme {
   private int longueur, largeur;

   public Rectangle(int longueur, int largeur) {
      this.longueur = longueur;
      this.largeur = largeur;
   }

   public float perimetre() {
      return 2 * (longueur + largeur);
   }

   public float surface() {
      return longueur * largeur;
   }
}
```

```
class Carre extends Rectangle{
    int cote;
    public Carre(int cote){ super(cote, cote); }
}
```

Polymorphisme

```java
public class TestFormes {
   public static void main(String[] arg) {
      Cercle cercle = new Cercle(2);
      Rectangle rectangle = new Rectangle(2, 1);


      Forme[] tableau = new Forme[4];
      tableau[0] = new Rectangle(10,20);
      tableau[1] = new Cercle(15);
      tableau[2] = new Rectangle(5,30);
      tableau[3] = new Carre(10);
      for (int i = 0 ; i < tableau.length ; i++)
      {
        if (tableau[i] instanceof Cercle)
            System.out.println("element[" + i + "] est un cercle");
        if (tableau[i] instanceof Rectangle)
            System.out.println("element[" + i + "] est un rectangle");
        if (tableau[i] instanceof Carre)
            System.out.println("element[" + i + "] est un carre");
      }
   }
}
```

```
C:\Users                    >java TestFormes
element[0] est un rectangle
element[1] est un cercle
element[2] est un rectangle
element[3] est un rectangle
element[3] est un carre
```

L'opérateur ***instanceof*** peut être utilise pour tester l'appartenance à une classe.
le choix du code à exécuter (pour une méthode polymorphe) ne se fait pas statiquement à la compilation mais dynamiquement à l'exécution.

Polymorphisme

```
class Animal {
    String nom;
    Animal(String n){nom = n;}
    void seDeplacer() { System.out.println(nom+" se déplace"); }
    void sePresenter(){ System.out.println(nom+" est un animal");}
}

class Chien extends Animal{
    Chien(String nom)
    {
        super(nom);
    }
    void sePresenter(){ System.out.println(nom+" est un chien");}
    void aboyer(){ System.out.println("Haou Haou");}

}
```

```
public class MainAnimal {
    public static void main(String[] args){
        Animal chien1 = new Chien("Rex");
        Chien chien2 = new Chien("Bobi");

        chien1.seDeplacer();
        chien2.seDeplacer();

        chien1.sePresenter();
        chien2.sePresenter();

        //chien1.aboyer();
        ((Chien)chien1).aboyer();
        chien2.aboyer();

    }
}
```

Refusée en compilation

Il faut un transtypage explicite de la référence chien1 de type Animal.

Polymorphisme

- Transtypage implicite (sens fille ➔ mère)
  - Il est toujours possible d'utiliser une référence de la classe mère pour désigner un objet d'une classe dérivée (fille, petite-fille et toute la descendance).
  - Il y a alors transtypage implicite de la classe fille vers la classe mère.
  - Exemple: un Chien est un Animal; le contraire n’est pas toujours vrai.
- Transtypage explicite (sens mère ➔ fille)
  - Un transtypage explicite n'est pas toujours possible, même si les classes ont un lien de parenté. C'est au programmeur de vérifier que son transtypage n'engendrera pas d'erreur, c'est-à-dire que l'objet est bien effectivement du type dans lequel on transtype la référence.

Polymorphisme: Exercice

- On se propose de repérer les erreurs et de les corriger

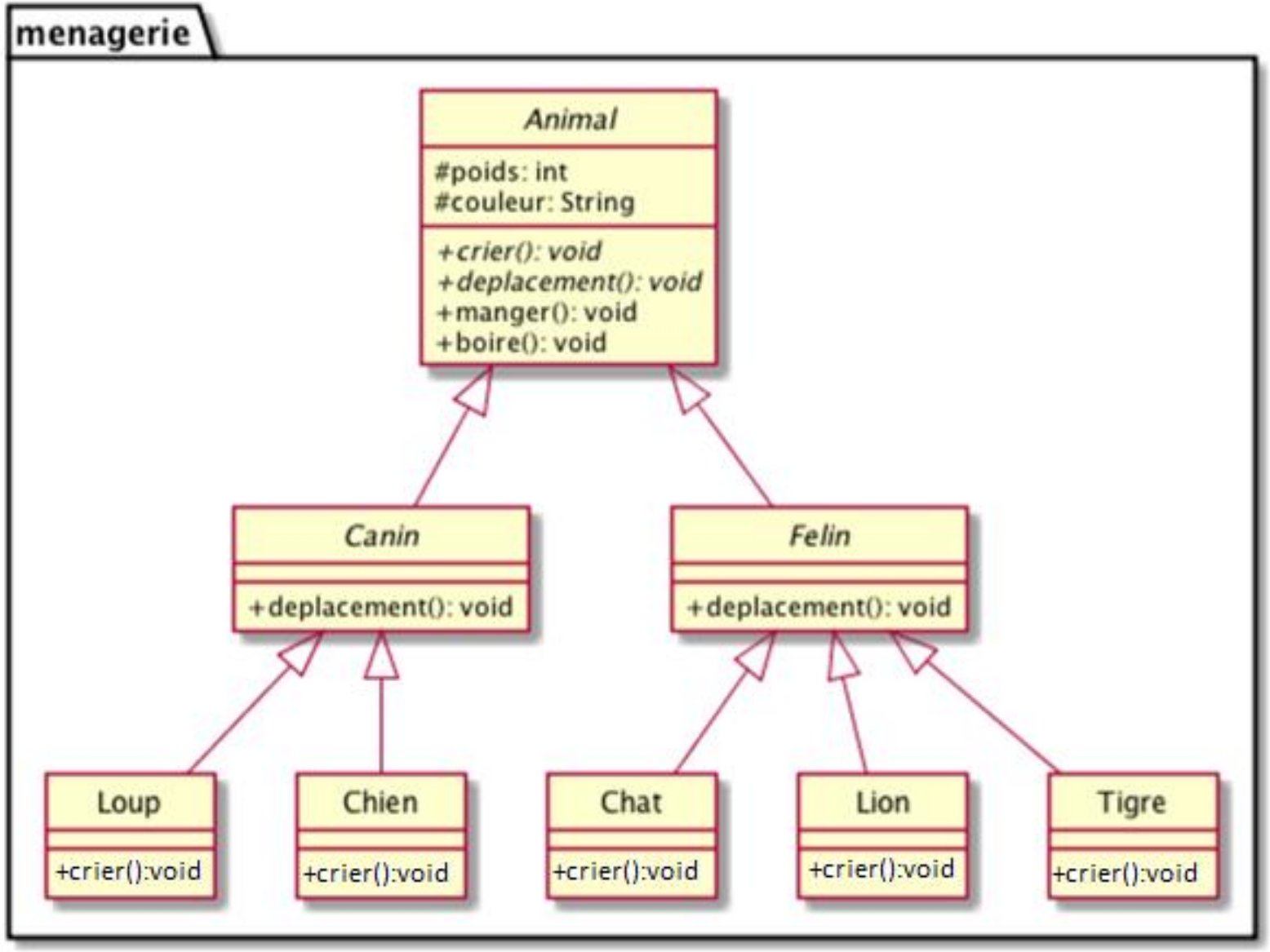
```
class Fruit
{
    public void methode1()
    {
        System.out.println("Je suis un fruit");
    }
}

class Pomme extends Fruit
{
    public void methode1()
    {
        System.out.println("Je suis une pomme");
    }

    public void methode2()
    {
        System.out.println("très délicieuse...");
    }
}
```

```
public class TestFruit
{
    public static void main(String args[])
    {
        Fruit monFruit = new Fruit();
        Fruit maPommeFruit = new Pomme();
        Pomme maPomme  = new Pomme();

        maPommeFruit.methode1();
        maPommeFruit.methode2();


        monFruit.methode1();

        monFruit = maPomme;

        monFruit.methode1();
        monFruit.methode2();


        maPomme = monFruit;

        maPomme.methode1();
        maPomme.methode2();
    }
}
```

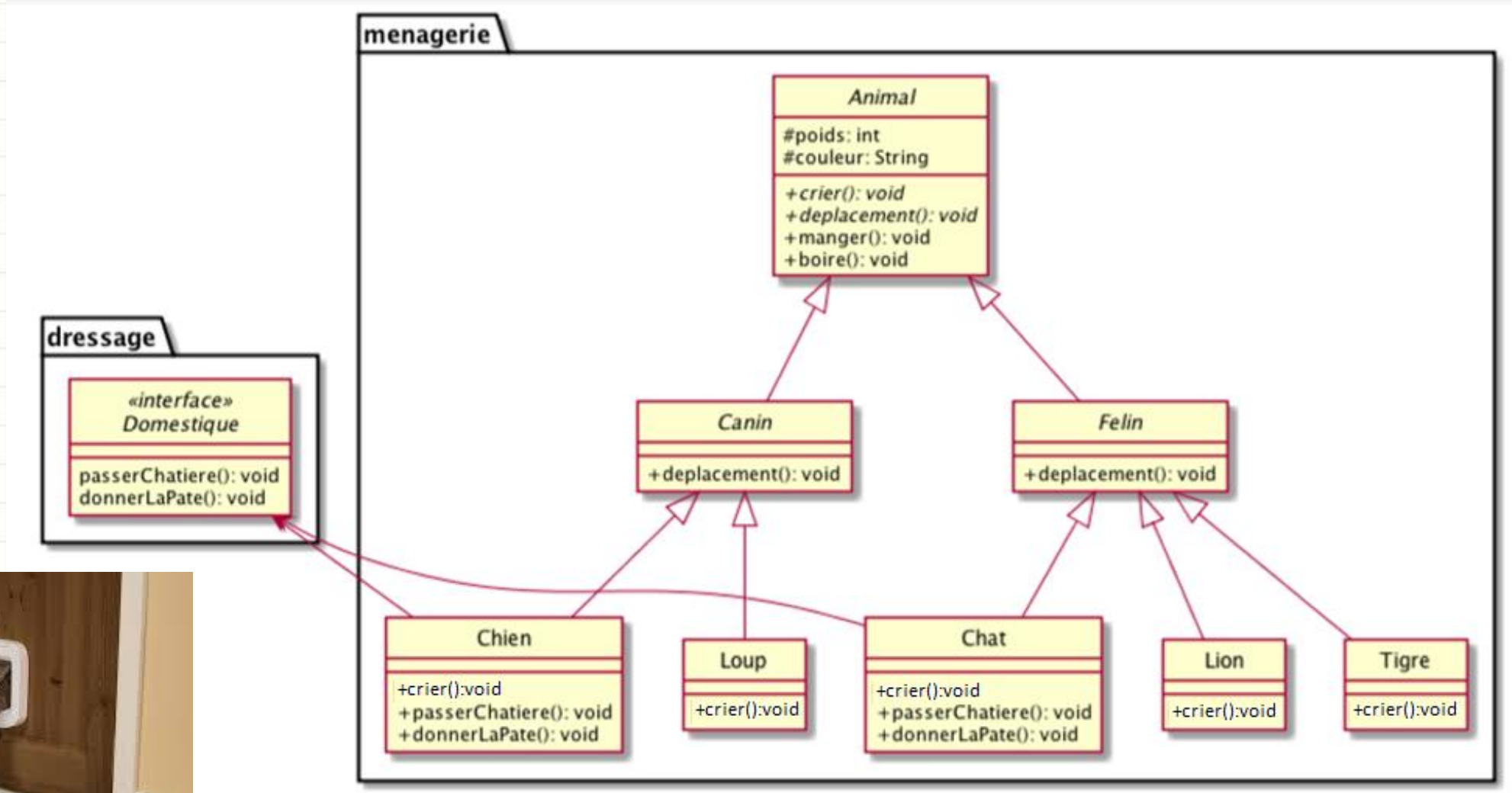
Polymorphisme: Exercice

```
public class CastFruit
{
    public static void main(String args[])
    {
        Fruit monFruit = new Fruit();
        Fruit maPommeFruit = new Pomme();
        Pomme maPomme  = new Pomme();

        maPommeFruit.methode1();
        ((Pomme)maPommeFruit).methode2();

        monFruit.methode1();              // va afficher : Je suis un fruit

        monFruit = maPomme;           // OK! - casting implicite normal
        monFruit.methode1();              // va afficher : Je suis une pomme

        //monFruit.methode2();            // ERREUR à la compilation : methode2()
                                          // n'existe pas dans la classe Fruit

        maPomme = (Pomme)monFruit;    // OK! - casting explicite et monFruit est
                                          // déjà une pomme
        maPomme.methode1();           // va afficher : Je suis une pomme
        maPomme.methode2();           // va afficher : très délicieuse...
    }
}
```

Représentation UML



Représentation UML



```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.");
    }
}
```

```java
package menagerie;

public abstract class Canin
    extends Animal {

    public void deplacement() {
        System.out.println("Je me
            déplace en meute.");
    }
}
```

```java
package menagerie;

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


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

}
```

```java
package dressage;

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

```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;
    }


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


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


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

}
```

```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;
    }


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


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


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

}
```

Agrégation

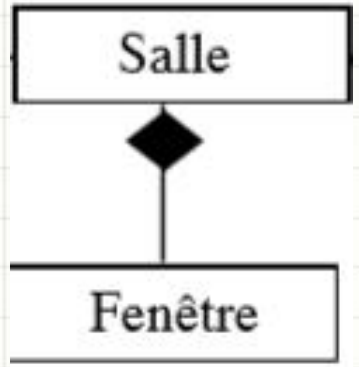
- L'agrégation est une association non symétrique dans laquelle une des extrémités, l'agrégat, joue un rôle prédominant par rapport à l'autre extrémité, l'élément agrégé.

- L'agrégation quant à elle est vue comme une relation de type **"a un"** ("has a"), c'est à dire que si un objet A a un objet B alors B peut vivre sans A.

Composition

- La composition peut être vue comme une relation **“fait partie de”** (“part of”), c’est à dire que si un objet B fait partie d’un objet A alors B ne peut pas exister sans A. Ainsi si A disparaît alors B également.

- La composition implique donc une coïncidence des durées de vie des composants et du composé.



Agrégation et composition: durée de vie des objets

- Techniquement, si le langage gère un « ramasse-miettes » (garbage collector), tout est composition !
- Quand on supprime un objet, les attributs sont supprimés.
- Si l'attribut est un autre objet ou une collection d'objets, ce sont des références vers l'objet attribut. Si l'objet attribut n'est plus référencé par personne, il sera détruit par le « ramasse-miettes ». Si l'objet attribut est référencé par une autre référence, il ne sera pas détruit par le « ramasse-miettes ».

Le ramasse-miettes « Garabage Collector »

- Un programme peut donner naissance à
- un objet en recourant à l'opérateur new.
- A sa rencontre, Java alloue un emplacement mémoire pour l'objet et l'initialise (par le constructeur).
- En revanche, il n'existe aucun opérateur permettant de détruire un objet dont on n'aurait plus besoin.
- La démarche employée par Java est un **mécanisme de gestion automatique de la mémoire** connu sous le nom de **ramasse-miettes (en anglais Garbage Collector)**.

Le ramasse-miettes « Garabage Collector »

- Son principe est le suivant :
  - A tout instant, on connaît le nombre de références à un objet donné. On notera que cela n'est possible que parce que Java gère toujours un objet par référence.
  - Lorsqu'il n'existe plus aucune référence sur un objet, on est certain que le programme ne pourra plus y accéder. Il est donc possible de libérer l'emplacement correspondant, qui pourra être utilisé pour autre chose. Cependant, pour des questions d'efficacité, Java n'impose pas que ce travail de récupération se fasse immédiatement. En fait, on dit que l'objet est devenu candidat au ramasse-miettes.

Le ramasse-miettes « Garabage Collector »

Remarque:

- On peut créer un objet sans en conserver la référence, comme dans cet exemple artificiel :

```
(new Point(3,5)).affiche() ;
```

- Ici, on crée un objet dont on affiche les coordonnées. Dès la fin de l'instruction, l'objet (qui n'est pas référencé) devient candidat au ramasse-miettes.