Héritage et polymorphisme
Classes abstraites et Interfaces

Stephane Malandain – POO - Java

Hes·SO GENÈVE
Haute Ecole Spécialisée
de Suisse occidentale

L'avenir est à créer

h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

I. Java et le mécanisme d'héritage

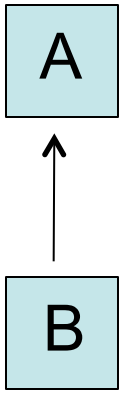
- **Concepts fondamentaux de POO** : modularité et hiérarchisation
- **Objectifs** : réutilisation et structuration du code
- **Modularité** : chaque composant du système est une unité fonctionnelle indépendante
- **Hiérarchisation** : concevoir des modules les plus abstraits possibles et les structurer en allant du général au particulier.

I. Java et le mécanisme d'héritage

- La classe « B » hérite de la classe « A »

```
class A {
    … // code de la classe A
    … // A est la super classe de B
    }

class B extends A {
    … // B hérite des propriétés de A
    … // et rajoute son propre code
    … // B est une sous classe de A
    }
```



Notation UML pour
B hérite de A

I. Java et le mécanisme d'héritage

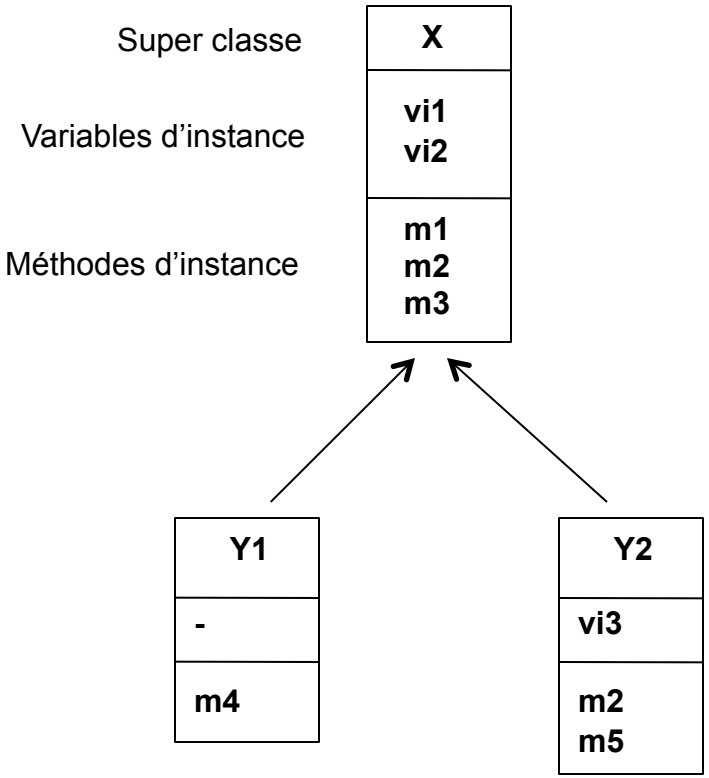
- **Héritage** : si B hérite de A, tous les membres (champs et méthodes) définis dans A deviennent des membres à part entière de B, et toute modification ultérieure de A sera automatiquement reportée sur B.

- **Héritage de la structure** : La classe B hérite de la structure et du comportement de A. La sous classe hérite des champs d'instance et de classe de sa super-classe. Possibilité de rajouter des champs propres mais pas d'en supprimer.

- **Terminologie** : B est une classe dérivée de A, ou encore une sous-classe de A. A est la super-classe de B.

I. Java et le mécanisme d'héritage

- Diagramme UML d'héritage de X par Y1 et Y2



I. Java et le mécanisme d'héritage

- **Héritage du comportement** : On peut ajouter de nouvelles méthodes et redéfinir les méthodes dont elle hérite.
- Ex . Classe `point` et `PointCol` (ajout d'un membre nommé `couleur` et d'une méthode `colore()`)

```
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 {
       public void colore(byte couleur) { this.couleur =
       couleur; }
   private byte couleur;
}
```

I. Java et le mécanisme d'héritage

- D'une manière générale, un objet d'une classe dérivée accède aux membres publics de sa classe de base exactement comme s'ils étaient définis dans la classe elle-même.
- Utilisation de l'exemple précédent :

```
…
PointCol pc;
PointCol pc2= new PointCol();
pc = new PointCol();
…
…
pc.affiche();
pc.initialise(3,5);
pc.colore((byte)3);
…
```

II. Accès d'une classe dérivée aux membres de sa classe de base.

- 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 **publics** de sa classe de base.

- Par exemple, pour afficher un objet de type `PointCol` avec ses coordonnées `x,y` et sa couleur on devra définir la méthode suivante :

```
…
public void affiche() {
     super.affiche();
     System.out.println(" et ma couleur est : " +
couleur);
}
…
```

III. Utilisation de l'héritage

- **Spécialisation** : le but d'une sous-classe sera de "spécialiser" sa super-classe en augmentant sa structure et son comportement. On obtient ainsi une hiérarchie de type "**généralisation – spécialisation**".

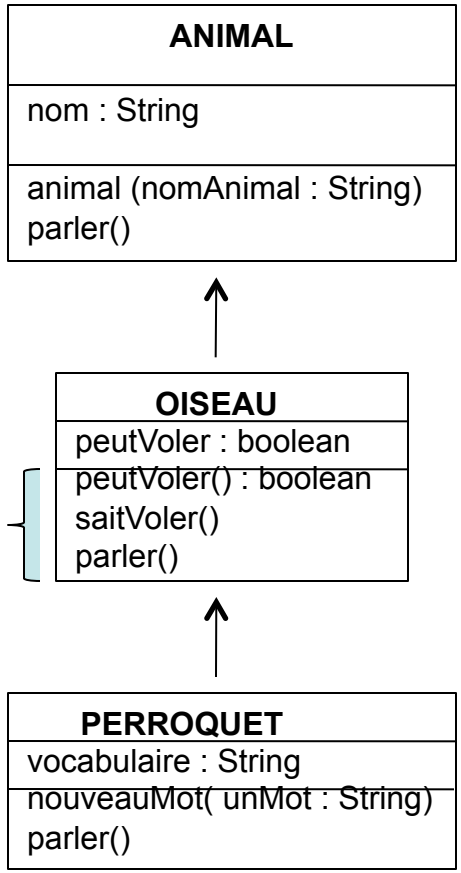
- **La relation** qui lie une sous-classe avec la super-classe dont elle hérite est une relation de type "*Est un*".

- **Exemple** : un perroquet est un oiseau, un oiseau est un animal.

- **Sous-typage** : l'héritage permet de créer des "sous-types" d'un type de base.
  - Types de base : éléments généraux partagés par l'ensemble des sous-types.
  - Sous-types : attributs supplémentaires et fonctionnalités enrichies par de nouvelles méthodes. Eventuellement, redéfinition de certaines méthodes pour tenir compte de spécificités particulières.

III. Utilisation de l'héritage

- Exemple



OISEAU spécialise ANIMAL en ajoutant les méthodes ci-contre

OISEAU redéfinit le comportement de ANIMAL en redéfinissant la méthode "parler"

IV. Construction d'objets dérivés

- 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, désigné par le mot clé **"super".**

```
class Point {
    public Point(int x, int y) { this.x=x; this.y=y; }
    …
}

class PointCol extends Point {
      public PointCol(int x, inty, byte couleur) {
        super(x,y); // première instruction OBLIGATOIRE
        this.couleur = couleur;
      }
      …
```

IV. Construction d'objets dérivés

- Remarques :
  - Appel d'un autre constructeur avec le mot clé `this` incompatible avec le mot clé `super`. Pourquoi ?
  - Une même classe peut avoir plusieurs classes dérivées différentes.
  - Java n'accepte pas l'héritage **multiple** (par contre, notion d'interface permet de simuler l'héritage multiple sans les inconvénients de ce dernier).

V. Redéfinitions et surdéfinitions de membres

- La **surdéfinition** (= surcharge) d'une méthode permet de cumuler plusieurs méthodes de même nom.
- La **redéfinition** d'une méthode substitue une méthode à une autre (cf. polymorphisme).
- Une classe dérivée peut fournir une nouvelle définition d'une méthode d'une classe ascendante.
- Méthode de même nom, de même signature et de même type de valeur de retour.
- Ex.: nouvelle définition de la méthode `affiche`() à la place de `affiche`() en utilisant le mot clé `super` (sinon `affiche`() serait récursive).

```
   …
public void affiche(){
      super.affiche();
      System.out.println (" et de couleur : " + couleur);
}
```

V. Redéfinitions et surdéfinitions de membres

```
class A {
     public void f (int n) { … }
     …
     }
Class B extends A {
     public void f (float x) { … }
     …
     }
     A a; B b;
     int n; float x;
     …
     a.f(n);      -> appelle f(int) de A
     a.f(x);      -> erreur de compilation

     b.f(n);      -> appelle f(int) de A
     b.f(x);      -> appelle f(float) de B
```

- Remarque : la recherche d'une méthode acceptable ne se fait qu'en remontant la hiérarchie d'héritage

V. Redéfinitions et surdéfinitions de membres

- Remarque : surdéfinition et redéfinition peuvent naturellement cohabiter, mais cela peut conduire à des situations très complexes qu'il est préférable d'éviter en soignant la conception des classes.

- Cas particulier : valeurs de retour covariantes. La nouvelle méthode peut renvoyer une valeur de type identique **ou dérivée** de celui de la méthode qu'elle redéfinie.

```
class A {
     public A f () { … }
     …
     }
Class B extends A {
     public B f () { … } // B.f redéfinit bien A.f
     …
     }
```

V. Redéfinitions et surdéfinitions de membres

- Les bonnes redéfinitions
  - Une redéfinition ne devrait pas modifier le comportement logique d'une méthode.
  - Une redéfinition peut éventuellement enrichir ce comportement, mais pas l'appauvrir.
- Redéfinitions et mode de protection
  - Seules les méthodes de mode d'accès "`public`" ou "`protected`" peuvent être redéfinies avec au moins la même visibilité que la méthode redéfinie.
  - Une méthode privée peut par contre être redéclarée au niveau d'une sous-classe et sera complétement indépendante de la première.

VI. Méthodes et classes finales

- Méthodes "finales"
  - Méthode publique ou protégée dont la redéfinition est interdite aux sous-classes.

```
final public void m() { … }
```

  - Sécurité (comportement d'une telle méthode ne sera pas modifié quelque soit le type courant de l'objet qui reçoit le message).
  - Efficacité (recherche de la bonne implémentation inutile dans le cas des méthodes finales)

- Classes "finales" `final class Xxx { … }`
  - Une classe finale ne peut avoir de sous-classes. Toutes ses méthodes sont donc implicitement finales.

V. Le polymorphisme

- Permet de manipuler des objets sans en connaître exactement le type.
- Permet d'écrire des blocs d'instructions indépendants du type des objets manipulés.
- Le type courant de l'objet désigné par une variable ou un paramètre de méthode peut prendre plusieurs formes. Elle n'est connue qu'à l'exécution du programme.
- Ex1 : classe "**`ArrayList`**" en java. Certaines méthodes ont des arguments de type "`Object`" qui peuvent donc prendre n'importe quelle forme à l'exécution.
- Ex 2 : Méthode `affiche()` dans classes `Point` et `PointCol`

V. Le polymorphisme

- Considérons la situation suivante :

```
Class Point {
    public Point(int x, int y) { … }
    public void affiche () { … }
}

Class PointCol extends Point {
        public PointCol(int x, inty, byte couleur) { … }
        public void affiche () { … }
    }
```

- Avec les instructions :

```
Point p;
p = new Point (3,5);
```

- Il se trouve que Java autorise le genre d'affectation suivante :

```
p = new PointCol (4,2, (byte)2);
```

V. Le polymorphisme

- Java permet d'affecter une référence à un objet du type correspondant mais aussi une référence à un objet du type dérivé.
- => conversion implicite (légale) d'une référence à un type classe T en une référence à un type ascendant de T.
- Considérons maintenant les instructions suivantes :

```
Point p = new Point (3,5);
p.affiche();
p = new PointCol (4,2, (byte)2);
p.affiche();
```

- `p` est de type `Point` alors que l'objet référencé est de type `PointCol`. La dernière instruction `p.affiche()` appelle la méthode `affiche()` de la classe `PointCol`. Elle se fonde 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 choix d'une méthode au moment de l'exécution porte le nom de **ligature dynamique.**

V. Le polymorphisme

- Le polymorphisme en Java se traduit donc par :
  - **La compatibilité par affectation entre un type classe et un type ascendant.**
  - **La ligature dynamique des méthodes.**
- Il permet d'obtenir un comportement adapté à chaque type d'objet sans avoir besoin de tester sa nature de quelque façon que ce soit.
- L'instruction `switch` est à la POO ce que l'instruction `goto` est programmation procédurale. (= le bon usage de la POO permet d'éviter des instructions de test).

VI. La super-classe Object

- Il existe une classe nommée ***Object*** dont dérive implicitement toute classe simple.

```
Class Point extends Object {
    …
}
```

- Une variable de type `Object` peut être utilisée pour référencer un objet de type quelconque. Utile pour manipuler des objets dont on ne connait pas le type exact.
- Exemple :

```
Point p=new Point (…) ;
Object o;
…
o = p;
o.deplace();                    // erreur de compilation, méthode
                                 // deplace n'existe pas dans Object
((Point)o).deplace();           // OK en compilation
Point p1 = (Point) o;           // OK, idem ci-dessus, avec création
p1.deplace();                   // d'une référence intermédiaire dans p1
```

VI. La super-classe Object

- La classe `Object` dispose de méthodes qu'on peut soit utiliser telles quelles, soit redéfinir.
- La méthode ***toString***
  - Renvoie une chaîne (objet de type *String*).
  - Contient le nom de la classe concernée et l'adresse de l'objet en hexadécimal (précédé de @)
  - Possible évidemment de redéfinir la méthode à notre convenance.
- La méthode ***equals***
  - Se contente de comparer les adresses des 2 objets concernés.
  - Possible également de redéfinir la méthode à notre convenance.

```
class Point { …
   boolean equals (Point p) { return ((p.x==x) && (p.y==y)) ; }
}

Point a = new Point(1,2);
Point b = new Point(1,2); // a.equals(b) => true
Object o1 = new Point(1,2);
Object o2 = new Point(1,2); // o1.equals(o2) => false
```

VII. Méthodes et classes abstraites

- Une classe abstraite est une classe qui contient au moins une méthode abstraite.

```
// Une classe abstraite
public abstract class Xxx {

    // méthode abstraite, sans instruction,
    // devant être implémentée par une classe descendante
    public abstract void m1();

    // méthode "concrète", avec instructions,
    // qui peut être redéfinie par une classe descendante
    public void m2() {
          <instructions de m2>
    }
}
```

- Une méthode abstraite est une méthode qui n'a pas de corps d'instructions.

VII. Méthodes et classes abstraites

- Une classe abstraite contient un mélange de méthodes abstraites et de méthodes concrètes (ou "normales").
- Une classe abstraite ne peut pas être instanciée.
- Elle sert uniquement à définir un cadre général (ensemble des caractéristiques communes aux sous-classes qui en seront dérivées).

```
// suite exemple précédent
Xxx unObjet;

unObjet = new Xxx(); // REFUSé PAR LE COMPILATEUR
```

- Pour créer l'objet "`unObjet`" il faut :
  - Définir une nouvelle classe qui étende la classe abstraite par héritage et ne soit plus abstraite
  - Cette sous-classe doit reprendre chacune des méthodes abstraites et leur proposer une implémentation.

VII. Méthodes et classes abstraites

- Exemple :

```
// concrétisation de la classe "Xxx"
public class Yyy extends Xxx {
    …
    // concrétisation de la méthode "m1"
    public void m1() {
        <instructions de m1>
    }

    // une méthode propre a "Yyy"
    public void m3() {
        <instructions de m3>
    }
}
```

VII. Méthodes et classes abstraites

- Soit les instructions :

```
Xxx o1; // ok. Le futur objet de type déclaré Xxx pourra répondre
aux messages "m1", "m2", mais pas "m3".

o1 = new Yyy(); // "Yyy" est le type courant de "o1"

Yyy o2 = new Yyy(); // "Yyy" est le type déclaré de "o2"

o1.m1();  // OK -> méthode concrète de "Yyy" invoquée
o1.m2();  // OK -> méthode concrète "m2" de "Xxx" invoquée
o1.m3();  // REFUSE PAR LE COMPILATEUR
o2.m1();  // OK -> méthode concrète de "Yyy" invoquée
o2.m2();  // OK -> méthode concrète de "Xxx" invoquée
o2.m3();  // OK -> méthode de "m3" de "Yyy" invoquée
```

- Une classe qui ne contiendrait que des méthodes abstraites et qui n'aurait aucune variable d'instance correspond plutôt à une interface et aurait du être déclarée comme telle.

VIII. Les interfaces

- Une **interface** = une classe abstraite qui implémente aucune méthode et aucun champ (hormis les constantes)

- Une interface définit un/le comportement qui doit être implémenté par une classe, sans implémenter celui-ci. C'est un ensemble de méthodes abstraites et de constantes.

- Notion plus riche que la classe abstraite car :

  - Une classe peut implémenter plusieurs interfaces
  - La notion d'interface se superpose à celle de dérivation
  - Les interfaces peuvent donc se dériver
  - On peut utiliser des variables de types interface

VIII. Les interfaces

- Liste de méthodes dont on donne seulement la signature. Attention, ce n'est pas une classe !
- Représente un "contrat", ce qu'on attend d'un objet
- Peut être implémentée par une ou plusieurs classes qui doivent donner une implémentation pour chacune des méthodes annoncées (et éventuellement d'autres).
- Une classe peut implémenter plusieurs interfaces (permettant un héritage multiple).

VIII. Les interfaces

- Toutes les méthodes d'une interface sont implicitement publiques et abstraites.
- Une interface n'a pas de constructeurs
- Une interface ne peut avoir de champs sauf si ceux-ci sont statiques.
- Une interface peut être étendue par une ou plusieurs autre(s) interface(s).

VIII. Les interfaces

- Exemple 1 :

```
// Définition de l'interface

public interface Vehicule {

   void rouler();
   void freiner();

}
```

- On a défini ici ce qu'on attend d'un objet de type `Vehicule`.

- On peut maintenant donner une ou plusieurs implémentations de cette interface grâce à l'utilisation du mot clef **implements**.

VIII. Les interfaces

```
public class Velo implements Vehicule {

   private String marque;
   private int rayonRoue;

   public Velo(String marque, int rayonRoue)
         this.marque=marque;
         this.rayonRoue=rayonRoue;
   }

   // méthodes

   public void rouler() {
         // coder ici la manière dont le vélo roule
   }

   public void freiner() {
         // coder ici la manière dont le vélo freine
   }

   // Autres méthodes propres à Velo
}
```

VIII. Les interfaces

```
public class Auto implements Vehicule {

   private String marque;
   private int poids;

   public Auto(String marque, int poids)
         this.marque=marque;
         this.poids=poids;
   }

   // méthodes

   public void rouler() {
         // coder ici la manière dont l'auto roule
   }

   public void freiner() {
         // coder ici la manière dont l'auto freine
   }

   // Autres méthodes propres à Auto
}
```

VIII. Les interfaces

- Dans cet exemple, nous avons donné 2 implémentations de `Vehicule`
- Ces 2 objets peuvent dons être vus comme des véhicules, c'est la base du **polymorphisme**
- Partout ou on attend un objet de type `Vehicule`, on peut mettre ces 2 objets
- On introduit ainsi une couche d'abstraction dans la programmation qui devient beaucoup plus flexible

VIII. Les interfaces

- Si nous avons une classe `Personne` possédant une méthode `conduire(Vehicule v)`, on peut alors écrire :

```
Personne p = new Personne();

p.conduire(new Velo());
// la méthode attend un Vehicule en argument, on peut donc passer tout
objet implémentant cette interface

p.conduire(new Auto()); // idem

// on peut instancier un Vehicule par le biais de ses inplémentations

Vehicule a = new Auto();
Vehicule v = new Velo();

// dans ce cas, a et v sont de type Vehicule, on ne peut appeler sur
ces objets que les méthodes définies dans l'interface Vehicule.
```

VIII. Les interfaces

- Exemple 2 :

```
public interface Animal
{
   // tous les animaux doivent implémenter les méthodes suivantes
   void manger(String nourriture);
   void seDeplacer();
   void respirer();
}

public interface Parler
{
   // une interface plus spécifique
   void parle(int phrase);
}

public interface Aboyer
{
   void aboie();
}
```

VIII. Les interfaces

```
public class Homme implements Animal, Parler
{
   //implémentations de l'interface Animal
   public void manger(String nourriture)
   {
      System.out.println("Miam, " +nourriture+ "!");
   }
   public void seDeplacer()
   {
      System.out.println("déplacement de l'homme");
   }
   public void respirer()
   {
      System.out.println("respiration de l'homme");
   }

   //implémentations de l'interface Parler
   public void parle(int phrase)
   {
      System.out.println(phrase);
   }
}
```

VIII. Les interfaces

```
public class Chien implements Animal, Aboyer
{
   //implémentations de l'interface Animal
   public void manger(String patee)
   {
      System.out.println("Miam, " +patee+ "!");
   }
   public void seDeplacer()
   {
      System.out.println("déplacement du chien");
   }
   public void respirer()
   {
      System.out.println("respiration du chien");
   }

   //implémentations de l'interface Aboyer
   public void aboie()
   {
     System.out.println("Ouaf!");
   }
}
```

VIII. Les interfaces

- Exemple d'interface dans l'API Java :
  - `Collection, List, Queue, Set, SortedSet`
  - `Cloneable, Comparable, Serialisable, Closeable, Runnable`
  - `ComponentListener, MouseListener, XxxxListener, …`

VIII. Les classes anonymes

- Java permet de définir ponctuellement une classe, sans lui donner de nom (essentiellement pour faciliter la gestion des événements)

```
// On dispose d'une classe A

A a;
a = new A() { // Champs et méthodes que l'on introduit dans
              // dans la classe anonyme dérivées de A
            } ;
…
```

- D'une manière générale, une classe anonyme ne s'applique que dans 2 cas :
  - Elle est dérivée d'une autre
  - Elle implémente une interface.

VIII. Les classes anonymes

- Exemple cas 2 :

```
Interface Affichable {
   public void affiche();
}

public class Anonym2 {
   public static void main (String[] args) {
         Affichable a;
         a = new Affichable() {
            public void affiche() {
                  System.out.println("Je suis un anonyme
                                    implementant Affichable");
            }
         };
         a.affiche();
   }
}
```

- On peut aussi transmettre la référence d'une classe anonyme en argument d'une méthode ou en valeur de retour