Chapitre 4:

Des concepts avancés de la programmation orientée objet Java

Héritage, polymorphisme, classes abstraites, interfaces, …

Plan du chapitre

- Paquetages
- Réutilisation
  - Par composition
  - Par extension
- Héritage
- Classes abstraites
- Polymorphisme
- Interfaces

Les paquetages

- Un paquetage regroupe des classes traitant de la même catégorie de fonctionnalités.
- Elle est proche de la notion de bibliothèque que l'on rencontre dans d'autres langages
  - Par exemple **java.io** qui regroupe tout ce qui concerne les entrées/sorties.
- Les classes d'un paquetage sont dans un même répertoire.
  - A chaque classe son fichier, à chaque package son répertoire
  - `java.awt.Point -> java/awt/Point.class`

- Dans un fichier AAA.java la déclaration package exemple au début du texte source indique que la classe contenue dans ce fichier fait partie du paquetage exemple.

Les paquetages

- En l'absence d'instruction package dans un fichier source, le compilateur considère que les classes correspondantes appartiennent au paquetage par défaut (situé dans le répertoire courant).
- Les paquetages étant organisés en arborescences, la désignation d'un paquetage s'effectue en donnant le chemin sous la forme pointée.
- pack.sous-pack.exemple désigne le paquetage exemple est situé dans le paquetage sous-pack, lui même situé dans le paquetage pack.
- Physiquement il s'agit du répertoire :

  ./pack/sous-pack/exemple

Les paquetages

- Si la classe **Point** appartient au paquetage **MesClasses,** on peut l'utilisez simplement en la nommant **MesClasses.Point**

```
MesClasses.Point p = new MesClasses.Point(2,5);
...
p.affiche(); // ici, le nom de paquetage n'est pas requis
```

  - Cette démarche devient fastidieuse dès que de nombreuses classes sont concernées
- L'instruction **import** permet d'utiliser une ou plusieurs classes, par exemple :

```
import MesClasses.Point, MesClasses.Cercle;
```

  - A partir de là, on peut utiliser les classes Point et Cercle sans avoir à mentionner à chaque fois leur nom de paquetage
- **import MesClasses.*;** permet d'utiliser toutes les classes du paquetage **MesClasses** en omettant le nom de paquetage correspondant à chaque appel.

Les paquetages

- Il existe un paquetage particulier nommé java.lang qui est automatiquement importé par le compilateur. C'est ce qui permet d'utiliser des classes standard telles que Math, System, Float, Integer, sans introduire d'instruction import
- Chaque classe dispose de ce qu'on nomme un droit d'accès. Il permet de décider quelles sont les autres classes qui peuvent l'utiliser. Il est simplement défini par la présence ou l'absence du mot clé public :
  - Avec public, la classe est accessible à toutes les autres classes (moyennant éventuellement le recours à une instruction import)
  - Sans le mot clé public, la classe n'est accessible qu'aux classes du même paquetage
- L'absence de mot clé (private ou public) pour un membre d'une classe veut dire que l'accès au membre est limité aux classes du même paquetage (accès de paquetage)

```
import java.util.Vector; // en début de fichier
// ...
Vector unVecteur;
```

équivaut à :

```
java.util.Vector unVecteur;
```

La ligne suivante permet d'utiliser toutes les classes du paquetage java.util

```
import java.util.*;     // en début de fichier
```

```
package P1;
public class A // accessible partout
{
……
void f1();
public void f2() {…..}
}
```

```
package P2;
Import P1.A;
class B // accessible que de P2
{
……
public void g();
{
A a;
a.f1(); //interdit
a.f2(); // OK
}
}
```

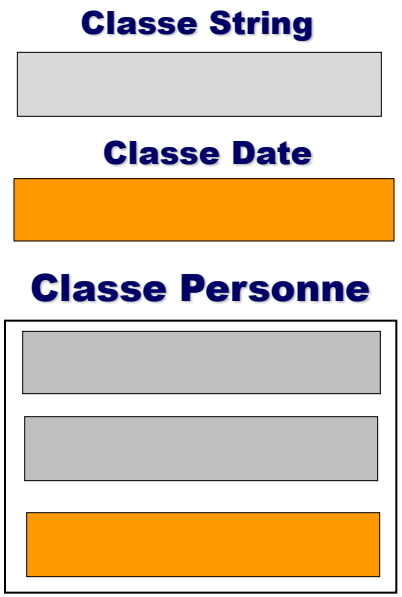
Ne confondez pas le droit d'accès à une classe avec le droit d'accès à un membre d'une classe

Reparlons de réutilisation

- Réutilisation par composition
  - Quand une classe est testée et elle est opérationnelle on peut l'utiliser aussi en tant que nouveau type dans une autre classe.
    - Les données d'une classe peuvent être des instances d'autres classes.

```
Class Date
{
    .....
}
```

```
class Personne
 {
        private String nom;
        private String prénom;
        private Date naissance;
        .....
}
```



Reparlons de réutilisation

```
class Date {
        private int j;          // jour
        private int m;          // mois
        private int a;          // an

        Date( )
        { j=1 ; m=1 ; a=1900 ; }

        Date(int j, int m, int a)
        { this.j = j ; this.m = m ; this.a = a ; }

        void affiche( )
         {  System.out.println(j + "/" + m +  "/" + a) ;  }
}
```

```
class Personne {
        private String nom;
        private String prénom;
        private Date naissance;
        Personne(String nom, String prénom, int
            jour, int mois, int an)
        {   this.nom=nom;
            this.prénom=prénom;
            naissance=new Date(jour, mois, an);
        }
        void affiche()
        {
            System.out.println("Identité Personne : ");
            System.out.println(nom+ " " +prénom);
            naissance.affiche();
        }
}
```

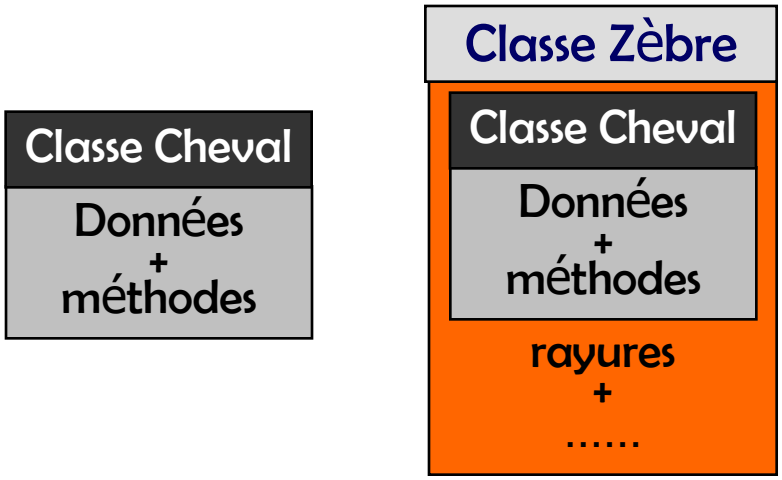
```
class ApplicationPersonne {
        public static void main(String args[ ])
        {
            Personne p = new Personne("Jacques", "DUPONT",
            1,2,1947);
            p.affiche();
        }
}
```

Résultat d'affichage

```
Identité Personne :
Jacques DUPONT
1/2/1947
```

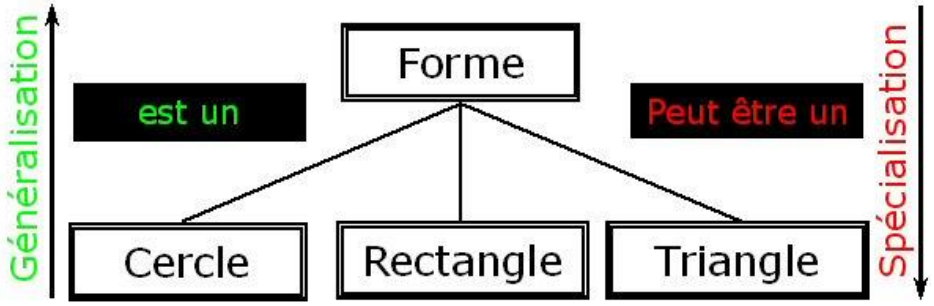
Reparlons de réutilisation

- Réutilisation par extension : héritage
  - Une classe représentant un concept voisin d'une classe existante n'a pas besoin d'être définie à partir de rien.
    - définition d'une classe comme extension d'une autre.
    - spécification des modifications seulement.
    - La deuxième classe hérite de la première.
    - classe initiale = classe mère ou de base ou superclasse
    - deuxième classe = classe dérivée (héritière) de la première ou sous-classe



Héritage

- **Java implémente le mécanisme d'héritage simple qui permet de "factoriser" de l'information dans le cas où deux classes sont reliées par une relation de généralisation / spécialisation.**



- **Pour le programmeur, il s'agit d'indiquer, dans la sous-classe, le nom de la superclasse dont elle hérite en utilisant le mot réservé : extends**
- **Exemple : class Cercle extends Forme**

Héritage

- L'héritage supprime, en grande partie, les redondances dans le code.
- on utilise l'héritage lorsqu'on définit un objet, par exemple Etudiant qui «est un» autre objet de type par exemple Personne avec plus de fonctionnalités qui sont liés au fait que l'objet soit un étudiant
- l'héritage permet de réutiliser dans la classe Etudiant le code de la classe Personne sans toucher au code initial : on a seulement besoin du code compilé
- l'héritage minimise les modifications à effectuer : on indique seulement ce qui a changé dans Etudiant par rapport au code de Personne, on peut par exemple
  - ✓ rajouter de nouvelles variables
  - ✓ rajouter de nouvelles méthodes
  - ✓ modifier certaines méthodes

Héritage

- Par défaut, une classe (qui n' a pas d'extends dans sa définition) hérite de la classe **Object (java.lang.Object)** qui est la superclasse de toutes les classes
- Une référence d'une classe C peut contenir des instances de C ou des classes dérivées de C.
- le mot clé super désigne la superclasse
- Une classe ne peut hériter que d'une seule classe et n'a donc qu'une seule classe mère
- la classe qui hérite ne peut pas retirer une variable ou une méthode
- Il est possible d'interdire qu'une méthode soit re-définie dans une sous-classe en utilisant final.
- Il est possible d'interdire qu'une classe puisse être héritée en utilisant final

Héritage et constructeurs

- Par défaut le constructeur d'une sous-classe appelle le constructeur "par défaut" (celui qui ne reçoit pas de paramètres) de la superclasse.
- Pour forcer l'appel d'un constructeur précis, on utilisera le mot réservé **super.** Cet appel devra être **obligatoirement** la **première instruction** du constructeur.
- Si la classe dérivée ne possède aucun constructeur alors son constructeur par défaut appelle un constructeur sans argument de la classe de base. Ce qui signifie que la classe de base devra :
  - Soit posséder un constructeur public sans argument, lequel sera alors appelé
  - Soit ne posséder aucun constructeur, il y aura appel du constructeur par défaut

Héritage et constructeurs

1. class A {
... // aucun constructeur
}
class B extends A {
public B (...) {
super(); // appelle le constructeur par défaut de A
..... }
}

Il reste permis de ne doter la classe B d'aucun constructeur

2. class A {
public A() {.....} // constructeur 1 de A
public A (int n) {.....} // constructeur 2 de A
}
class B extends A { ..... // pas de constructeur }

B b = new B(); // construction de b → appel de constructeur 1 de A

Héritage et constructeurs

3. class A {
   public A (int n) {……} // constructeur 2 seulement
}
class B extends A {
 …… // pas de constructeur
}

Erreur de compilation car le constructeur par défaut de B cherche à appeler un constructeur sans argument de A.

4. class A {
…….. // pas de constructeur
}
class B extends A {
…… // pas de constructeur
}

Aucun problème ! La création d'un objet de type B entraîne l'appel de constructeur par défaut de B, qui appelle le constructeur par défaut de A

Héritage et constructeurs

- **Situation usuelle :** la classe de base et la classe dérivée disposent toutes les deux d'au moins un constructeur public.

```
public class Employe extends Personne
{
public int nbheures;
public Employe (String nom,
                String prenom,
                 int anNaissance,
                  int nbheures)
 {
  super(nom, prenom, anNaissance);
  this.nbheures = nbheures;
 }
}
```

```
public class Personne
{
  public String nom, prenom;
  public int anNaissance;
  public Personne()
  {
    nom=""; prenom="";
  }
  public Personne(String nom,
                  String prenom,
                  int anNaissance)
  {
    this.nom=nom;
    this.prenom=prenom;
    this.anNaissance=anNaissance;
  }
}
```

Héritage et masquage

- Une classe peut définir des variables portant le même nom que celles de ses superclasses
- Une classe peut accéder aux attributs redéfinis de sa classe mère en utilisant super
- Une classe peut accéder aux méthodes redéfinies de sa classe mère en utilisant super.
- L'opérateur instanceof permet de savoir si une instance est instance d'une classe donnée.
  - Renvoie une valeur booléenne
- Une référence d'une classe C peut contenir des instances de C ou des classes dérivées de C

Héritage et masquage

```
class A {
        int x;
        void m() {...}
}
```

```
class B extends A{
        int x;
        void m() {...}
}
```

```
class C extends B {
    int x, a;
    final void m() {...}
    void test() {
            a = super.x; // a reçoit la valeur de la variable x de la classe B
            a = super.super.x; // erreur syntaxique
            a = ((B)this).x;   // a reçoit la valeur de la variable x de la classe B
            a = ((A)this).x;   // a reçoit la valeur de la variable x de la classe A
            super.m();         // Appel à la méthode m de la classe B
            super.super.m();   // erreur syntaxique
            ((B)this).m();     // Appel à la méthode m de la classe C (et non B)
    }
}
```

Héritage et masquage

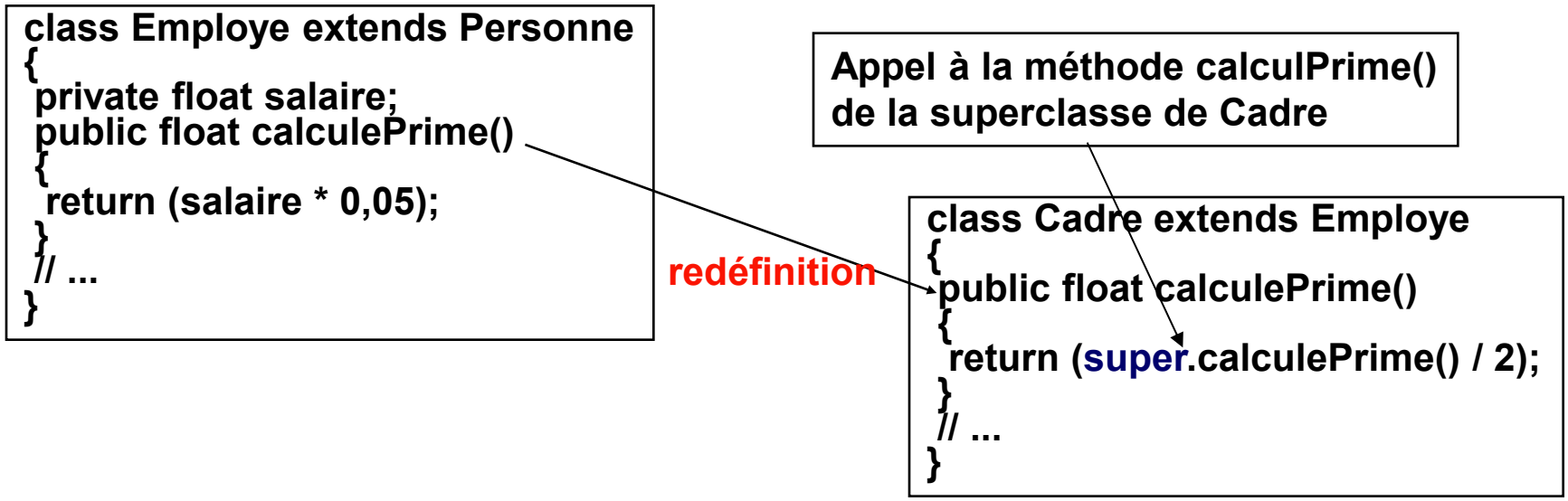
```
public class Ellipse {
        public double r1, r2;
        public Ellipse(double r1, double r2) {
              this.r1 = r1;
              this.r2 = r2;
   }
        public double area{...}
}
```

```
final class Circle extends Ellipse {
        public Circle(double r) {
              super(r, r);
        }
        public double getRadius()
        {return r1;}
}
```

```
Ellipse e = new Ellipse(2.0, 4.0);
Circle c = new Circle(2.0);
System.out.println("Aire de c:" + c.area());
System.out.println(e instanceof  Circle);  // false
System.out.println(e instanceof  Ellipse); // true
System.out.println(c instanceof  Circle);  // true
System.out.println(c instanceof  Ellipse); // true
e = c;
System.out.println(e instanceof  Circle);  // true
System.out.println(e instanceof  Ellipse); // true
double r = e.getRadius(); // Erreur
c = e; // Erreur type incompatible
```

Héritage et masquage

- Une sous-classe peut re-définir des méthodes existant dans sa superclasse, à des fins de spécialisation.
  - On parle aussi de **masquage**.
  - La méthode redéfinie **doit avoir la même signature**



Héritage et masquage

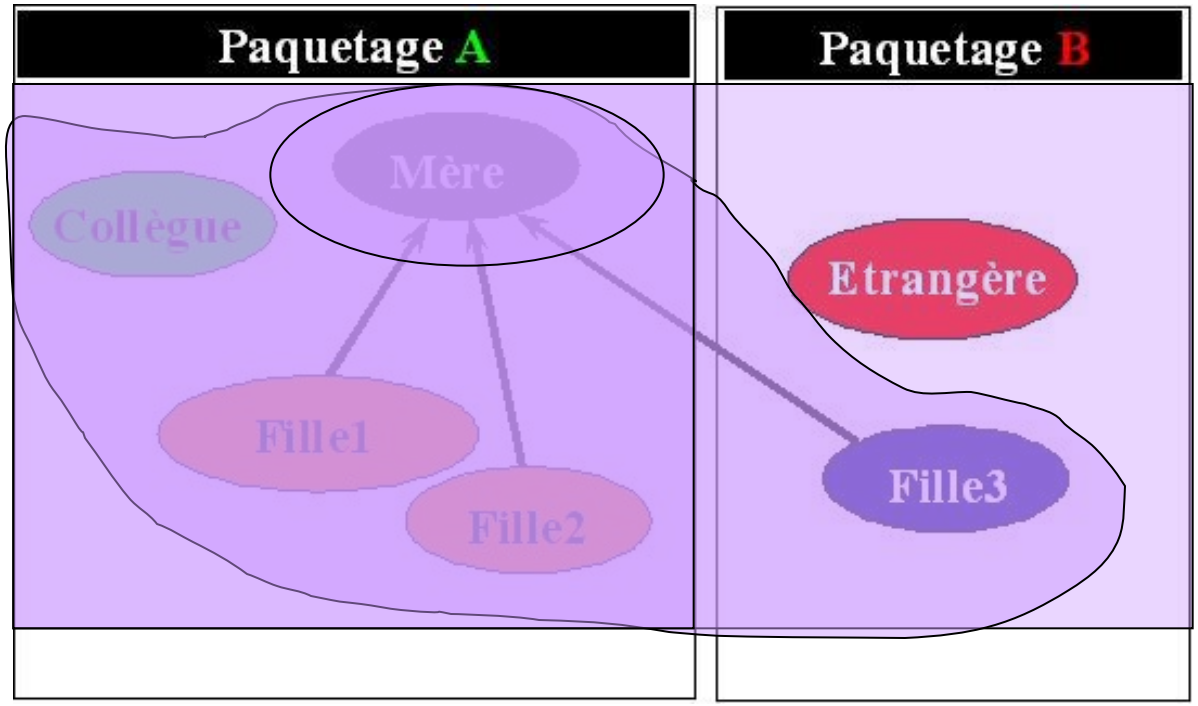
- Lorsque l'on redéfinit une méthode cela signifie
  - que l'on modifie son implémentation
  - mais que l'on ne change pas son interface
  - pour appeler une méthode de la super-classe, on doit précéder le nom de la méthode par le mot clé super : super.getNom()
  - lorsque l'on redéfinit une méthode on ne peut pas réduire le niveau d'accès de la méthode, c'est-à-dire on ne peut pas passer d'une méthode public à private
- Une classe fille peut *surcharger* une méthode
  - ce qui signifie qu'elle peut réécrire une méthode en gardant le même nom mais pas la même signature
    - c'est-à dire avec des paramètres d'entrée différents mais même paramètre de retour
  - cela signifie que l'on ajoute un comportement en modifinat son interface

Héritage et masquage

- Obligation de redéfinir les méthodes déclarées comme abstraites (*abstract*)
- Interdiction de redéfinir les méthode déclarées comme finales (*final*)
- une classe déclarée final ne peut pas avoir de classes dérivées
- Interdiction d'accès aux attributs privés de la classe mère
  - nécessité d'utiliser des fonctions d'accès publiques
  - Le modificateur de visibilité protected permet l'accès des sous-classes aux membres de la classe de base.

Héritage et visibilité

- Private
- Public
- Protected
- Aucun modificateur de visibilité



Héritage et visibilité

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Source: https://www.geeksforgeeks.org/java/protected-vs-private-access-modifiers-in-java/

Exemple

```
class Compteur {
        private  int valeur ;
        Compteur() { valeur = 0 ; }
        void inc( ) {valeur ++ ; }
        void dec( ) {valeur --; }
        int vaut() { return valeur ; }
}
```

```
class Compteur_Controle extends Compteur
{
        private int maxval;
        Compteur_Controle(int maxval)
        { super();
        } this.maxval = maxval;
        void inc() {

        }  if( vaut( ) < maxval ) super.inc( );
        int get_maxval( ) { return maxval; }
}
```

```
class TestControle_Compteur {
{
        public static void main(String args[])
        {
            Compteur_Controle c = new Compteur_Controle(6);
            c.inc( );                    // Première incrémentation
            c.inc( );                    // Deuxième Incrémentation
            System.out.println("valeur : "+c.vaut( ) );
            c.dec( );                    // Première décrémentation
            c.dec( );                    // Deuxième décrémentation
            System.out.println("valeur : "+c.vaut( ) );
        }
}
```

Classes abstraites

- Il peut être nécessaire au programmeur de créer une classe déclarant une méthode sans la définir (c'est-à-dire sans en donner le code). La définition du code est dans ce cas laissée aux sous-classes : classe abstraite
- Une classe abstraite est introduite par le mot clé « abstract » et elle a au moins une méthode abstraite.
- Une méthode abstraite est déclarée par le mot clé abstract et elle n'a pas de corps.
- Une classe abstraite ne peut pas être instanciée (new).
- Si une classe dérivée d'une classe abstraite ne redéfinie pas toute les méthodes abstraites alors elle est abstraite.
- Pour utiliser un classe abstraite on doit définir une classe héritière qui fournit les réalisations des méthodes abstraites de la classe abstraite.

Classes abstraites

- Méthode abstraite : méthode n'admettant pas d'implémentation.
  - au niveau de la classe dans laquelle elle est déclarée abstraite, on ne sait pas la réaliser.
- Les méthodes dites " static ", " private " et " final " ne peuvent pas être abstraites
  - Static : méthode de classe
  - Private : méthode interne à cette classe, non héritable, donc on ne peut pas la redéfinir.
  - Final : aucune surcharge de la méthode possible.
- Une classe déclarée final ne peut pas être abstraite
- Une classe déclarée abstraite est abstraite même si aucune méthode abstraite n'y figure. Il faut en hériter pour l'instancier

Classes abstraites

```
class abstract Shape
{
…
   public Point getPosition() {
                return posn;
                                }

   public abstract double perimeter();
…
}
```

```
class Circle extends Shape
{
    …
    public double perimeter() { return 2 * Math.PI * r ; }
}
```

```
class Rectangle extends Shape
{
    …public double perimeter() { return 2 * (height + width); }
}
```

Polymorphisme

- Le polymorphisme est le fait que la même opération puisse se comporter différemment sur différentes classes de la hiérarchie.
- "*Le polymorphisme constitue la troisième caractéristique essentielle d'un langage orienté objet après l'abstraction des données (encapsulation) et l'héritage*" Bruce Eckel "Thinking in JAVA"
- Des classes différentes peuvent avoir des méthodes différentes de même nom et de même signature
- Le polymorphisme consiste pour *une instance* à retrouver quelle méthode doit être appelée.

```
Point3D unPoint3D = new Point3D( x, y, z );
Point unPoint     = (Point) unPoint3D;
unPoint.projete(); // Point3D.projete() ou Point.projete() ?
```

Polymorphisme

- Le polymorphisme permet la manipulation uniforme d'objets de plusieurs classes via une classe de base commune.
  - Plus besoin de distinguer différents cas en fonction de la classe des objets.
- Si on définit dans une classe C (sous-classe de B) une méthode m et avec la même signature que la méthode m de la classe B alors cette méthode m de la classe B est masquée pour toutes les instances de C.
- On ne peut pas appeler directement la méthode m de B à partir d'une référence de C.
- Exemple :

```
Public class C extends B {...}
C unC = new C();
B unB = unC; // Transtypage
unB.m(); //c'est la méthode m() de C si elle existe sinon de B !
```

Polymorphisme

- Il ne faut pas confondre :
  - le type de la référence de l'objet (ce qui apparaît dans le code)
  - le type de l'objet effectivement utilisé (celui qui est construit)
- Le premier est vérifié à la compilation
- Le second est vérifié à l'exécution.
- Il faut que les deux soient compatibles
- Pour que l'on puisse compiler et exécuter, il faut que le type effectif hérite du type déclaré
- Exemple :

class PointCol extends Point {…}

```
Point p = new PointCol();
System.out.println(p.getName());
```

=> Affiche PointCol et non pas Point

```
Point p1 = new Point(); // OK
Point p2 = new PointCol();// OK
PointCol p3 = new PointCol(); // OK
PointCol p4 = new Point();
// erreur compilation
```

Polymorphisme

- Soit la classe Etudiant qui hérite de la classe Personne
  - *Soit une méthode getNom() de Personne qui est redéfinie dans Etudiant*
  - quelle méthode getNom() sera exécutée dans le code suivant, celle de Personne ou celle de Etudiant?

```
Personne a =new Etudiant(5); // a est un objet de la classe
   a.getNom();                // Etudiant mais il est déclaré
                              // de la classe Personne
```

  - la méthode appelée ne dépend que du type réel (Etudiant) de l'objet a et pas du type déclaré (ici Personne)
  - c'est la méthode de la classe Etudiant qui sera exécutée

Polymorphisme

- Même code d'invocation de **`perimeter()`**
  - toujours sur un objet déclaré de type **`Shape`**

```
System.out.println(shapes[i].perimeter());
```

- appliqué aux objets de types différents…
  - on a un effet différent selon l'objet qui reçoit le message, et plus précisément selon sa classe

```
Shape[ ] shapes = {  new Circle(2),
                     new  Rectangle(2,3),
                     ……..
                   };
```

```
double  sum_of_perimeters = 0;

for (int i = 0; i < shapes.length; i++)

        sum_of_perimeters += shapes[i].perimeter( );
```

Exemple

```
public class Point {
  public Point(double x,double y){…}
  public double getX() {…}
  public double getY() {…}
  public void set(double x,double y)
{…}
  public String toString() {…}
};
```

```
public class Polar extends Point {
  public Polar(double x,double y) {…}
  public void set(double x,double y)
{…}
  public String toString() {…}
  public double getR() {…}
  public double getTheta() {…}
};
```

```
Point m = new Polar(3,4);
m.set(1,0);
System.out.println(m.getR());
```

refusé à la compilation!!!

Une référence à un `Point` ne peut pas accéder à une méthode de `Polar` même si elle référence en fait un `Polar`

```
Point m = new Point(3,4);

Polar p = (Polar) m;
System.out.println(p.getR());
```

erreur à l'exécution!!!

`p` ne référence pas une instance de type `Polar` donc on ne peut pas faire de *cast*.

```
Point m = new Polar(3,4);
m.set(1,0);

Polar p = (Polar) m;
System.out.println(p.getR());
```

Interfaces

- L'interface d'une classe = la liste des messages disponibles = signature des méthodes de la classe
- Certaines classes sont conçues pour ne contenir précisément que la signature de leurs méthodes, sans corps. Ces classes ne contiennent donc que leur interface, c'est pourquoi on les appelle elles-mêmes *interface*
- Ne contient que la déclaration de méthodes, sans définition (corps)
- Permettre une certaine forme de multi-héritage

Interfaces

- Une interface correspond à une classe où toutes les méthodes sont abstraites.
- Le modificateur abstract est facultatif
- Une classe peut implémenter (implements) une ou plusieurs interfaces tout en héritant (extends) d'une classe.
- Comme pour les classes, l'héritage est possible entre les interfaces
- Une classe peut implémenter plusieurs interfaces différentes
- la classe qui réalise une interface s'engage à implémenter les méthodes définies dans cette interface

Interfaces

- Jusqu'à **Java 7**, les interfaces **ne pouvaient contenir que** :
  - des **méthodes abstraites** (sans corps),
  - et des **constantes** (public static final).
- Mais à partir de **Java 8**, les interfaces peuvent contenir :
  - **Méthodes par défaut** (default): avec une **implémentation**.

```
interface Vehicule {
    default void demarrer() {
        System.out.println("Le véhicule démarre");
    }
}
```

  - **Méthodes statiques:** également avec une **implémentation**.

```
interface Outils {
    static void afficherInfo() {
        System.out.println("Interface d'outils");
    }
}
```

- Les classes qui implémentent l'interface peuvent :
  - **utiliser directement** la méthode default, ou
  - **la redéfinir** si elles veulent changer le comportement.

Interfaces

- **Depuis Java 9:** les interfaces peuvent aussi contenir :
  - **Méthodes privées** (utilisées pour factoriser du code interne à l'interface).

```
interface Calculatrice {
    private void log(String message) {
        System.out.println("Log: " + message);
    }

    default void addition(int a, int b) {
        log("Addition appelée");
        System.out.println(a + b);
    }
}
```

Version Java	Méthodes autorisées dans une interface
≤ Java 7	Abstraites seulement
Java 8	Abstraites, default, static
Java 9+	Abstraites, default, static et private

Interfaces

```
class abstract Shape { public abstract double perimeter(); }

interface Drawable { public void draw(); }

class Circle extends Shape implements Drawable {
    public double perimeter() { return 2 * Math.PI * r ; }
    public void draw() {...}
}
class Rectangle extends Shape implements Drawable {
    ...
    public double perimeter() { return 2 * (height + width); }
    public void draw() {...}
}
...
Drawable[] drawables = {new Circle(2), new Rectangle(2,3), new Circle(5)};
for(int i=0; i<drawables.length; i++)
        drawables[i].draw();
```

Interfaces

- Une interface est introduite par le mot clé « interface », et se comporte comme une classe dont toutes les méthodes sont abstract et dont tous les attributs sont final.
- Les mots clés final et abstract peuvent ne pas apparaître dans la définition d'une interface.
- Une interface est un moyen de préciser les services qu'une classe peut rendre. C'est un modèle d'implémentation.
- Une interface est inutilisable en elle-même.
- Une classe doit implémenter l'interface, c'est a dire définir les corps des méthodes abstraites de l'interface.
- Une interface est utilisée lorsque nous avons seulement besoin de savoir que telle classe possède telles méthodes et se comporte de telle manière

Interfaces

- Une classe peut implémenter plusieurs interfaces.
  - Elle doit dans ce cas fournir des définitions pour toutes les méthodes spécifiées dans ces interfaces.

```
interface A
{
   void e( );
}
```

```
interface B
{
   void f( );
}
```

```
class C implements A, B
{
   void e(  ) {- - -; }
   void f( ) {- - -; }
}
```

- Plusieurs classes différentes peuvent implémenter de manières différentes une même interface.
  - Plusieurs réalisations possibles d'une même spécification (polymorphisme).

Interfaces et redéfinition

- Lors d'un héritage multiple d'interfaces :

```
interface X {- - - -}
interface Y {- - - -}
interface Z extends X,Y {- - - -}
```

  - Il n'y pas de risque de duplication des attributs de X et de Y dans Z:
    - Les attributs sont final donc non héritables.
    - Et pour les méthodes ?

```
class Z implements X,Y {- - - -}
```

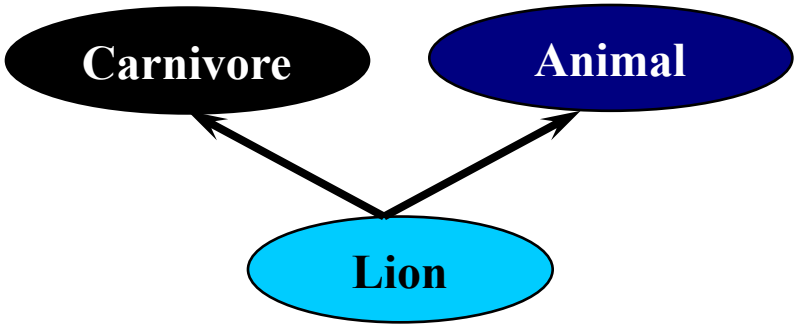
- Si dans les interfaces X et Y on trouve une fonction f avec les mêmes paramètres mais des types de retour différents.
    - void f(int i); // dans X
    - int f(int i); // dans Y
  - la classe Z ne peut pas implémenter les deux interfaces

Interfaces et redéfinition

- Si les méthodes f situées dans les interfaces X et Y ont les mêmes paramètre et le même type de retour.
  - int f(int i); // dans X
  - int f(int i); // dans Y
  - La classe Z aura une implémentation de f correspondant aux deux spécifications identiques.
- Si les méthodes f situées dans les interfaces X et Y ont des paramètres différents et n'importe quel type de retour.
  - void f(char c); // dans X
  - int f(int i); // dans Y
  - La classe Z aura deux implémentations de la méthode f qui correspond aux deux versions distinguées par les paramètres.

Interfaces et héritage multiple

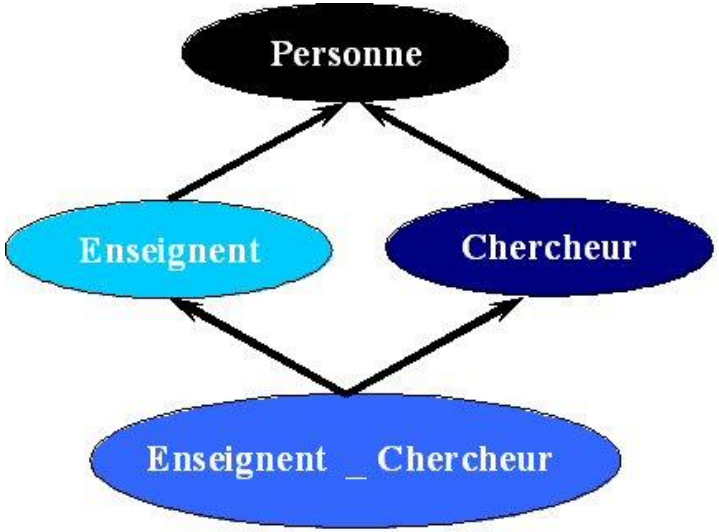
- Java ne permet pas l'héritage multiple :



- Peut-on dériver de deux classes? Non!
  - Pourquoi?
    - comment décider quels code appliquer par défaut si deux méthodes ont le même nom?
  - des solutions dans d'autres langages (C++,...)
  - pas en Java!
- Pourtant on en a besoin?!

Interfaces et héritage multiple

- La notion d'interface offre des possibilités de faire de l'héritage multiple
  - Les interfaces peuvent hériter de plusieurs autres interfaces
  - Une classe peut hériter d'une classe et implémenter plusieurs interfaces
- Comment résoudre ce problème (héritage en diamant) ?




```
interface Personne{- - -}
interface Chercheur extends Personne
    {- - -}
class Enseignant implements Personne
    {- - -}
class Enseignant_Chercheur extends
    Enseignant implements Chercheur
    {- - - }
```

Interfaces et héritage multiple

- On ne peut dériver que d'une classe
- On peut implémenter plusieurs interfaces
  - plus de problèmes de définition multiple
- Dans notre exemple :
  - Les attributs de Personne ne sont pas redéfinis dans Enseignant ni Chercheur
    - Ils sont déclarés final.
  - Les méthodes de Personne sont implémentées dans Enseignant, voir redéfinies dans Enseignant_Chercheur, mais ne peuvent pas être implémentées par Chercheur qui est une interface.