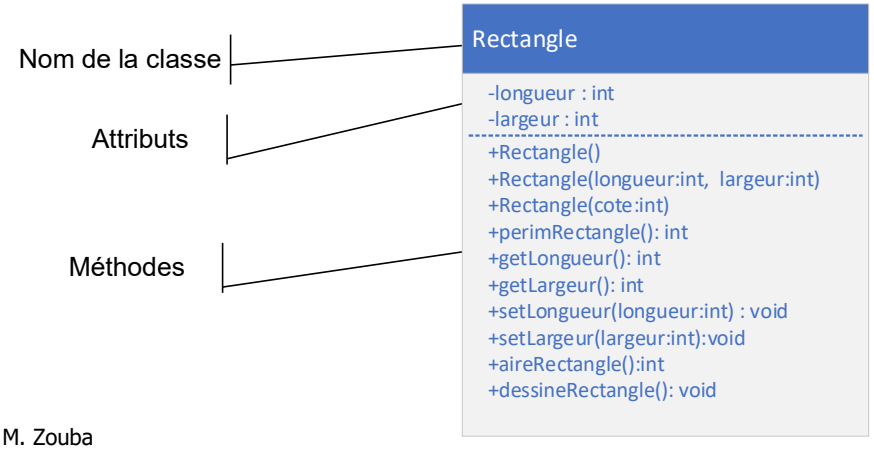
La relation de composition

Rappel: modélisation d'une classe

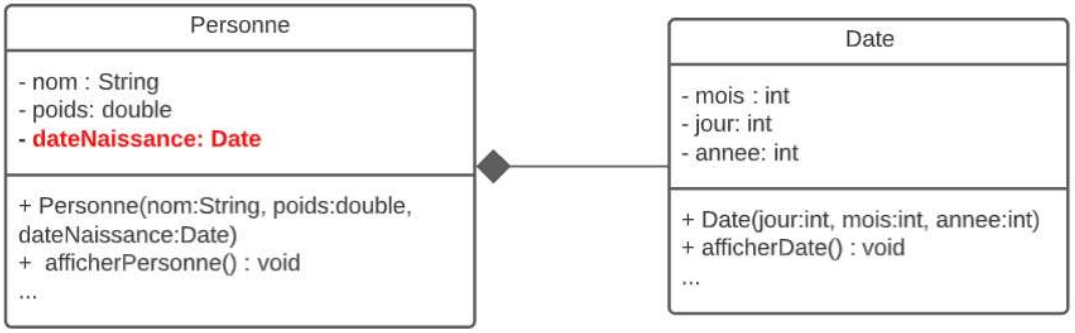
- Voici comment une classe peut être représentée en suivant les conventions de la modélisation objet **UML** (Unified Modeling Language).
- **On spécifie le nom de la classe, les attributs et les méthodes** dans un rectangle comme ici-bas.
- La visibilité des membres (attributs et méthodes) est représentée par les différents symboles suivants :
  - **private ou −** : le membre est accessible uniquement dans la classe en question.
  - **public ou +** : le membre est accessible dans toutes les autres classes du programme.
  - **protected ou #** : le membre est accessible uniquement dans les classes dérivées de la classe en question ainsi qu'aux classes se trouvant dans le même package.
  - **package ou ~ ou rien** : le membre est accessible uniquement dans les classes du même paquetage.



La relation de composition

- La composition représente une forme de réutilisation de logiciel qui permet à une classe de posséder comme attributs des références à des objets d'autres classes.
- C'est une relation «a un» ou «est composé de» qui signifie qu'un objet peut contenir un ou plusieurs objets d'autres classes.
- Exemples de relations de composition:
  - Dans le tp1, la classe `Compte` a un champ d'instance de la classe `Client`
  - Dans l'exemple de la semaine 3, la classe `Personne` a un champ d'instance de type `Date`
- La composition est indiquée par un lien débutant par un losange.

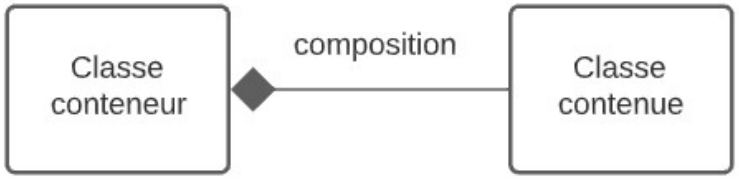
- Exemple: Une Personne a une Date




```
public class Personne {
    private String nom;
    private int poids;
    private Date dateNaissance;

    public Personne(String nom, int poids, dateNaissance Date) { … }
    …
}
public class Date {
    private int jour;
    private int mois;
    private int annee;
    public Date(int jour, int mois, int annee) {…}
    …
}
```

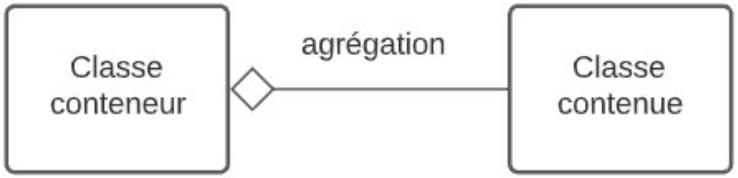
composition et agrégation

- Le **losange plein** représente la **composition** d'une classe. La classe contenue ne peut pas exister sans la classe conteneur,

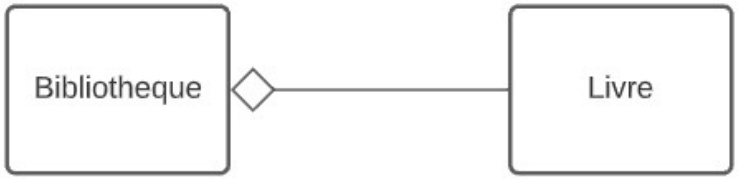


- Exemple: La classe `Train` est composée de la classe `Siege`. La classe `Siege` ne peut exister sans la classe `Train`.

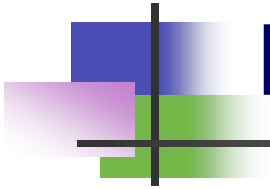
- Le **losange vide** représente **l'agrégation** d'une classe. La classe contenue peut exister sans la classe conteneur.



- Exemple: La classe `Bibiotheque` est composée de la classe `Livre`. La classe `Livre` peut exister sans la classe `Bibliotheque`.



La relation d'héritage



L'héritage

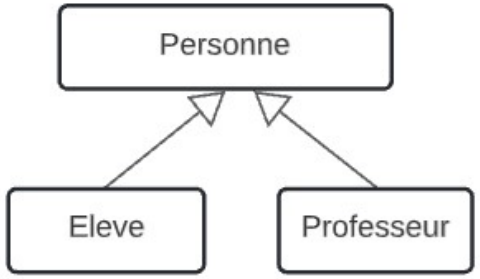
- L'héritage est une autre forme de réutilisation de logiciel qui consiste à créer de nouvelles classes à partir d'une autre classe déjà existante et ainsi récupérer les attributs et les méthodes, sans avoir à la réécrire complétement.
- Lorsqu'une classe hérite d'une autre classe, elle hérite (utilise) ses attributs et ses méthodes et on peut lui ajouter de nouveaux attributs et de nouvelles méthodes pour adapter la classe à la nouvelle situation (la spécialiser).
- Terminologie :
  - La classe définie à partir d'une autre classe est appelée la **classe fille** ou **sous-classe**. Celle qui sert à définir la classe fille est appelée la **classe mère** ou **super-classe**.
  - On dit que la classe fille **dérive (**ou **hérite** de ou **étend** la classe mère).

La relation d'héritage

- Exemples de relations d'héritage

SuperClasse	Sous-classe	Sous-classe
Personne	Eleve Professeur	
Forme	Quadrilatere Cercle Triangle	Rectangle Carre Losange
Prêt	PretAutomobile PretHypothcaire PretPresonnel	
Vehicule	Moto Velo Voiture Bateau	

- La relation d'héritage est une relation «**est un**» : chaque objet d'une sous classe est un objet de la super classe. Par exemple, un `Eleve` est une `Personne`.
- L'héritage est aussi une relation de **spécialisation**. un `Eleve` est une `Personne` particulière.
- L'ensemble de tous les objets de sous-classe (comme `Eleve`) est un sous ensemble de tous les objets de la super-classe(comme `Personne`).
- L'héritage est indiqué par une flèche au bout vide dirigée de la sous-classe vers la superclasse. La superclasse est au-dessus de la sous-classe.



L'héritage en Java

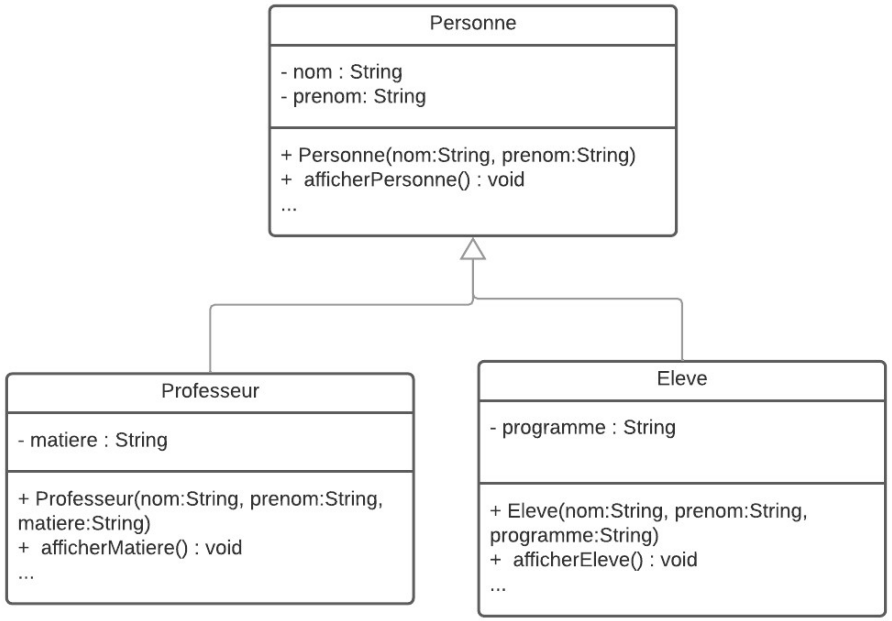
- On utilise le mot clé **extends** pour faire hériter une classe d'une superclasse.

```
Public class NomSousClasse extends NomSuperClasse {
      /* Déclaration des attributs et méthodes spécifiques à la
         sous-classe */
}
```

Exemple :

```
public class Professeur extends Personne {
      …
}
```

La relation d'héritage: exemple (1/3)



La relation d'héritage: exemple (2/3)

```
12      public Personne(String prenom, String nom) {
13          this.prenom = prenom;
14          this.nom = nom;
15      }
16
17      public String getPrenom() {
20
21      public void setPrenom(String prenom) {
24
25      public String getNom() {
28
29      public void setNom(String nom) {
32
33      public void afficherPersonne() {
34          System.out.println("Prenom:" + getPrenom());
35          System.out.println("Nom:" + getNom());
36      }
37
38      public void afficher() {
39          System.out.println("Renseignements de la personne.");
40          afficherPersonne();
41      }
42  }
```

```
11  public class Professeur extends Personne {
12      private String matiere;
13
14      public Professeur(String prenom, String nom, String matiere) {
15          // Appel du constructeur de la super classe Personne
16          super(prenom, nom);
17          this.matiere = matiere;
18      }
19
20      public String getMatieres() {
23
24      public void setMatieres(String matieres) {
27
28      public void afficherProfesseur() {
29          System.out.println("Renseignement de la personne.");
30          // Appel de la méthode afficherPersonne() de la superclasse Personne
31          super.afficherPersonne();
32          System.out.println("Matière:" + getMatieres());
33      }
34  }
```

La relation d'héritage: exemple (3/3)

```
10 public class Eleve extends Personne {
11     private String programme;
12
13     public Eleve(String prenom, String nom, String programme) {
14         super(prenom, nom);
15         this.programme = programme;
16
17     }
18
19     public String getProgramme() {
22
23     public void setProgramme(String programme) {
26     public void afficherEleve() {
27         System.out.println("Renseignements de l'élève:");
28         // Appel de la méthode afficherPersonne() de la superclasse Personne
29         super.afficherPersonne();
30         System.out.println("Programme: t" + getProgramme());
31     }
32
33 }
```

L'organisation des classes et l'héritage

- Quand on développe en orienté objet, on commence par concevoir les **classes les plus générales** que l'on **spécialise** au fur et à mesure en dérivant des **classes plus spécialisées.**
  - Exemple : on commence par concevoir la classe Personne (générale) ensuite les classes Professeur et Eleve (spécialisées)

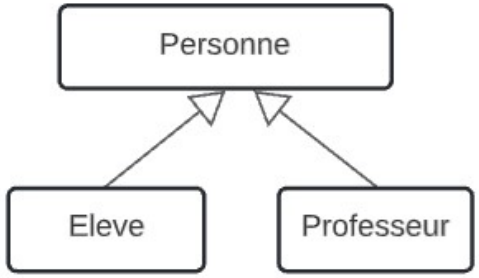
- Il faut organiser les classes de façon à ce que les propriétés et méthodes communes à plusieurs classes soient placées dans une superclasse.

- Une classe parent peut avoir plusieurs classes enfants.

- Cependant, une classe ne peut pas hériter de plusieurs classes en Java. L'héritage multiple en java n'est pas possible.

Particularisation-généralisation

- Particularisation-généralisation :
  - Un Professeur **est une** Personne, mais c'est une personne particulière.
  - Un Eleve est une Personne, mais c'est une personne particulière.



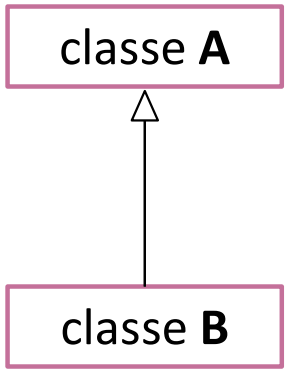
- On peut voir une Personne comme étant la **généralisation** de Eleve et de Professeur

- **Une classe enfant** (comme Eleve) **offre de nouveaux services** (méthodes). Elle peut redéfinir **les services rendus par la superclasse** (Personne).

Vocabulaire

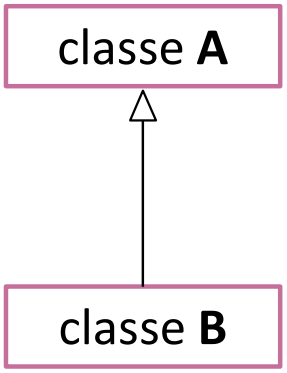
- On dit de la classe A:
  - La classe A est la superclasse de la classe B.
  - La classe A est la classe mère de la classe B.
  - La classe A est la classe parent de la classe B.

- On dit pour la classe B:
  - La classe B hérite de la classe A.
  - La classe B dérive de la classe A.
  - La classe B étend (extends) la classe A.
  - La classe B est une classe enfant de la classe A.
  - La classe B est une sous-classe de la classe A.



Réutilisation dans l'héritage

- Le type est hérité : un objet de la classe B est aussi un objet de classe A. La classe B hérite des attributs et des méthodes de la classe A , sauf les constructeurs.



- Qu'est ce qui change entre A et B ?
  - Le code source de B ne comporte que ce qui a changé par rapport au code de A.
  - On n'a pas besoin de recopier tout le code qui est dans A pour le mettre dans B.
- On peut, par exemple, faire les modifications suivantes:
  - Ajouter de nouvelles méthodes dans B .
  - Ajouter de nouveaux attributs dans B.
  - Modifier certaines méthodes de A qui se retrouvent ainsi dans B : d'autres fonctionnalités

Spécialisation

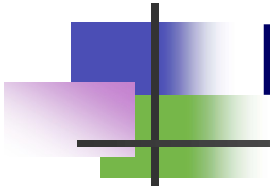
Enrichissement : redéfinition

Le mot-clé super

```
public class Personne {
   private String prenom;
   private String nom;

   public Personne(String prenom, String nom) {
      this.prenom = prenom;
      this.nom = nom;
   }
   ..
}
```

```
public class Professeur extends Personne {
   private String matiere;

   public Professeur(String prenom, String nom,  String matiere) {
   // Appel du constructeur de la super classe Personne
     super(prenom, nom);
     this.matiere = matiere;
   }
…
}
```

Attention: si on omet d'appeler le constructeur de la superclasse (dans la première ligne), le constructeur par défaut est appelé implicitement. Donc, il devrait être défini dans la superclasse sinon on aura un message d'erreur.



Le mot-clé super

- Le mot-clé ***super*** est utilisé pour désigner la **superclasse**.
- ***super*** permet d'accéder aux membres de la superclasse: ***super.nomMethode()*** de la même manière que l'on accède aux membres de la classe elle-même (comme le mot-clé ***this***).
- Le mot clé ***super*** indique que la recherche de la méthode doit commencer dans la superclasse plutôt que dans la classe courante.
- ***super()*** permet de faire appel au constructeur de la superclasse. Il permet de profiter du constructeur de la superclasse et évite la duplication de code.

Héritage et constructeur

- **Les constructeurs de la superclasse ne sont jamais hérités dans la sous-classe**, c'est pourquoi il faut les appeler à l'aide du mot-clé ***super***.

- La syntaxe pour appeler ce constructeur est ***super()*** ou ***super(****parametres****).***

- Cette instruction doit être **la première ligne** dans la définition du constructeur de la sous-classe.

- **Si *super()* n'est pas explicitement utilisé , alors le constructeur par défaut (constructeur sans paramètre) de la superclasse est appelé.**

Exemple 1

```
public class Forme {
     private int posX;
     private int posY;
     //constructeur par défaut
     public Forme() {
         posX = 0; posY = 0;}
     public Forme(int x, int y) {
          setPosX(x);
          setPosY(y);
     }
     public void setPosY(int y) { posY=y; }
     public void setPosX(int x) { posX=x; }
     public int getPosX() { return posX; }
     public int getPosY() { return posY; }
}
```

```
public class Rectangle extends Forme {
     private double largeur = 3.0;
     private double hauteur = 4.0;
     /*le constructeur par défaut fait appel au
     constructeur par défaut de la classe Forme. On
     a pu omettre le mot clé super car il le fera
     implicitement(pas conseillé)*/
     public Rectangle() {
          super();
          setHauteur(0);
          setLargeur(0);
     }
     /*le constructeur avec paramètres de la classe
     Rectangle fait appel au constructeur avec
     paramètres de la superclasse Forme en utilisant
     super(x,y)*/
     public Rectangle(double h, double l, int x, int
     y) {
          super(x,y);
          setHauteur (h);
          setLargeur(l);
     }
     public  double getLargeur() {return largeur;}
     public  double getHauteur() {return hauteur;}
     public void setLargeur(double l) {largeur=l;}
     public void setHauteur(double h) {hauteur=h;}
     }
```

Visibilité des membres

- Rappel : l'accès aux membres (attributs et méthodes) d'une classe pouvait être :
  - **public** : visibilité totale à l'intérieur et à l'extérieur de la classe (mot-clé public)
  - **privé** : visibilité uniquement à l'intérieur de la classe (mot-clé private)
  - **par défaut** (aucun modificateur) : visibilité depuis toutes les classes du même paquetage (est aussi valable pour le paquetage par défaut)

Visibilité dans le cas de l'héritage

- Il n'est pas possible d'utiliser un attribut ou méthode privés d'une superclasse dans la classe héritée.
- Et on ne peut pas les rendre public pour pouvoir les utiliser car c'est contre le principe de l'encapsulation.

- Solution : Modificateur protected

- Un attribut **protected** ou une méthode **protected** dans une classe **public**, peut être accessible dans *n'importe quelle classe du même package ou dans ses sous-classes*, même si les sous-classes sont dans des packages différents.

- Comparé au modificateur **public, protected** permet une meilleure encapsulation mais n'est pas aussi strict que **private**.

Exemple 2 : Modificateur protected

```
public class Forme {
    protected int posX;
    protected int posY;

    public Forme(int x, int y) {
        setPosX(x);
        setPosY(y);
    }
    //setter et getter de posX et posY

}
```

```
public class Rectangle extends Forme {
    private double largeur = 3.0;
    private double hauteur = 4.0;
    //....
    public Rectangle(double h, double l, int x, int
    y) {
        super(x,y);
        setHauteur(h);
        setLargeur(l);
    }
    //on ajoute la méthode coindDS à Rectangle
    public void coinDS() {
        System.out.println("Le coin sup droit est :
        " + (posX +largeur) + ", " + posY);
    }
    //setter et getter de largeur et  hauteur
}
```

Exemple 3

On a enrichi la classe Forme et la classe Rectangle
On a rajouté les classes Cercle et Cylindre

```
public class Forme {
     protected int posX=0;
     protected int posY=0;
     protected static int nbObjets = 0;


     public Forme(int x, int y){
          setPosX(x);
          setPosY(y);
          nbObjets++;
     }
     public void deplaceDe( int dx, int dy){
      setPosX(posX+dx); setPosY(posY+dy);
     }
     public static int getNbObjets() {
          return nbObjets;
     }
     //les getters et les setters …
}
```

```
public class Rectangle extends Forme {
     private double largeur = 3.0;
     private double hauteur = 4.0;
     public Rectangle(double h, double l, int x, int y) {
          super(x,y);
          setHauteur(h);
          setLargeur(l);
     }
     public void coinDS() {
          System.out.println("Le coin sup droit est : " +
          (posX + largeur) + ", " + posY);
     }
     public double surface() {
          return largeur*hauteur;
     }
     public double perimetre() {
          return (largeur+hauteur)*2;
     }
     public  double getLargeur() {return largeur;}
     public  double getHauteur() {return hauteur;}
     public void setLargeur(double l) {largeur=l;}
     public void setHauteur(double h) {hauteur=h;}
}
```

Spécialisation : rajout de méthodes coinDS(), surface() et perimetre().

```java
public class Cercle extends Forme {
    protected int rayon;
    public Cercle(int x, int y, int r){
        super(x, y);
        setRayon(r);
    }

    public void setRayon(int r)
     {
          rayon = r;
     }
     public double getRayon() {
          return rayon;
     }

     double perimetre(){
        return 2*Math.PI*rayon;
    }

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

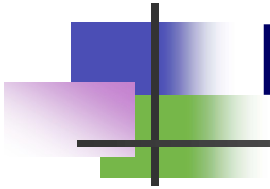
```java
public class Cylindre extends Cercle {
    private double longueur;

     public Cylindre(int x, int y, int rayon, double l)
     {
          super(x,y,rayon);  /* appel au constructeur
          de la super-classe */
          setLongueur(l);
     }

     public double getLongueur(){
          return longueur;
     }

     public void setLongueur(double longueur){
          this.longueur=longueur;
     }

     public double volume()
     {
          return super.surface() * longueur;
     }
}
```

```
public class TestAppelMethode {
public static void main(String arg[]) {
Rectangle rect1 = new Rectangle(5, 3, 2, 5);
Cercle cercle = new Cercle(4, 6, 12);
Cylindre monCylindre = new Cylindre(3, 5, 2, 10);
rect1.deplaceDe(1, 1);
cercle.deplaceDe(1, 1);
monCylindre.deplaceDe(1, 1);
System.out.println("**** les positions x et y après le déplacement **** ");
System.out.println("rectangle: x=" + rect1.getPosX() + "   y=" + rect1.getPosY());
System.out.println("cercle: x=" + cercle.getPosX() + "   y=" + cercle.getPosY());
System.out.println("cylindre: x=" + monCylindre.getPosX() + "   y=" +
monCylindre.getPosY());
rect1.coinDS();
System.out.println("le périmètre du cercle est=" + cercle.perimetre());
System.out.println("le volume du cylindre est=" + monCylindre.volume());
}
```

Appel aux méthodes héritées:
deplaceDe(int, int),
getPosX (),
getPosY()

Appel aux méthodes ajoutées:
perimetre(),
volume(),
coinDS()

```
**** les positions x et y après le déplacement ****
rectangle: x=3.0   y=6.0
cercle: x=5.0   y=7.0
cylindre: x=4.0   y=6.0
Le coin sup droit est : 3.0,6.03.0
le perimetre du cercle est=75.39822368615503
le volume du cylindre est=125.66370614359172
```



Redéfinition et surcharge de méthodes

- **La classe enfant peut :**
  - **Redéfinir** des méthodes de la classe parent (même signature).
  - **Surcharger** des méthodes de la même classe (même nom mais pas la même signature).

Méthodes redéfinies

- Une classe enfant peut **redéfinir** une méthode héritée en faveur de sa propre implémentation.
- La nouvelle méthode **doit** avoir la même signature que celle du parent mais elle peut avoir un **code différent.**
- Si la méthode à redéfinir de la superclasse est **public**, la méthode de redéfinition de la sous classe **doit** être aussi **public.**
- **Le type d'objet** qui exécute la méthode détermine la **version** de la méthode qui est invoquée, celle du parent ou de l'enfant.

Redéfinir = Override

Redéfinition, le mot clé super

- Une méthode de la classe parent peut être invoquée **explicitement** en utilisant la référence **super.**

- Pour appeler une méthode de la super-classe :

  La syntaxe est la suivante :
  super.nomMéthode(paramètres);

Exemple 4:

Dans la classe **Cylindre**, on redéfinie la méthode ***surface()***, car la surface d'un cylindre se calcule différemment de la surface d'un cercle.

```
class Cercle extends Forme {
    protected double rayon;
    Cercle(double x, double y, double r) {
        super(x, y);
        setRayon (r);
    }
    public void setRayon(double r)
     {
          rayon = r;
     }
     public double getRayon() {
          return rayon;
     }

     public double perimetre(){
        return 2*Math.PI*rayon;
    }
    public double surface(){
        return Math.PI*rayon*rayon;
    }
 }
```

Redéfinition

```
public class Cylindre extends Cercle {
    private double longueur; //rajout

     public Cylindre(int x, int y, int rayon, double l)
     {
          super(x,y, rayon);
          setLongueur(l);
     }

     public double getLongueur(){
          return longueur;
     }
     public double surface() { //redéfinition
          return 2 * super.surface() + super.perimetre()*
          longueur;
     }
     public double volume() //rajout
     {
          return super.surface() * longueur;
     }
}
```

Appel de la méthode surface de la superclasse Cercle en utilisant le mot clé super

Remarque

- Si, dans une super-classe et une sous-classe, il existe une méthode, par exemple la méthode ***surface()*** , ayant la même signature (même nom, même liste de paramètres) et qui retourne le même type de valeur, et si ***objet1*** est une instance de la super-classe et ***objet2***, une instance de la sous-classe, alors l'instruction suivante ***objet2. surface()*** va appeler la méthode déclarée dans la sous-classe.

- On dit que la méthode ***objet2. surface()*** redéfinit la méthode ***objet1. surface()***.

```
public class TestCylindre {
public static void main(String args[])
{
     Cylindre monCylindre = new Cylindre(3,5,2,10);

     System.out.println("La longueur est " +
     monCylindre.getLongueur());

     System.out.println("Le rayon est " +
     monCylindre.getRayon());

     System.out.println("Le volume du cylindre est " +
     monCylindre.calculerVolume());

     System.out.println("La surface du cercle est " +
     monCylindre.surface());

     System.out.println("Le nombre de cylindres est " +
     Cylindre.getNbObjets());
     }
}
```

La longueur est 10.0
Le rayon est 2.0
Le volume du cylindre est 125.66370614359172
La surface du cercle est 150.79644737231007
Le nombre de cylindres est 1

C'est la méthode ***surface()*** de la classe **Cylindre** qui est appelée

C'est la méthode ***getNbObjets()*** de la classe **Forme** qui est appelée

L'annotation @override

- L'annotation *@Override* indique au compilateur qu'une méthode donnée est la redéfinition d'une méthode de la superclasse.
- En plus de rappeler au programmeur qu'il s'agit de la redéfinition d'une méthode de la superclasse, la présence de l'annotation *@Override* va forcer le compilateur à vérifier que la méthode a été redéfinie correctement.

- Exemple dans la classe Cylindre :

```
@Override
public double surface() {
        return 2 * super.surface() + (2 * Math.PI * getRayon()) * longueur;
}
```

Surcharge vs. Redéfinition

- La **surcharge** concerne plusieurs méthodes avec le même nom dans la **même** classe mais avec **différentes signatures**.
  - La méthode *System.out.println()* est un bon exemple de la surcharge. On peut l'appeler avec différents types et nombres de paramètres.
  - La surcharge de méthodes permet de définir plusieurs constructeurs pour une même classe.
- La **redéfinition** concerne deux méthodes, l'une dans la classe parent et l'autre dans la classe enfant, ayant la même signature.

La surcharge permet de définir une opération de **manières différentes** pour différents paramètres.

La redéfinition permet de définir une **opération similaire de différentes manières** pour différents objets

Exemple de surcharge

```
public class Math {
    //…

    public static double carre(double x) {
        return x*x;
    }
    public static int carre(int x) {
        return x*x;
    }

    //…
}
```

Le modificateur final

- Le concepteur de classe peut imposer que la classe parent ne soit pas utilisable dans un héritage. Dans ce cas, on utilisera le mot clé final dans la déclaration de la classe.

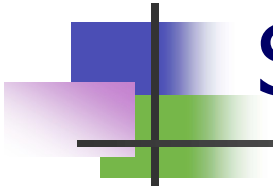
```
public final  class Cylindre extends Cercle {
…
}
public class Cone extends Cylindre {
…
}
```

The type Cone can't subclass the final class Cylindre

- On peut aussi définir une méthode comme étant **final**; une méthode finale ne peut pas être redéfinie par ses sous-classes.

```
public class Forme {
    protected int posX=0;
    protected int posY=0;
    public final void deplaceDe( int dx, int dy) {
        setPosX(posX+dx); setPosY(posY+dy);
    }
}
```

Les classes Rectangle, Cercle et Cylindre qui héritent de la classe Forme ne peuvent pas redéfinir la méthode deplaceDe(…)



Sources

- Kypriano, S. L'héritage
- Tasso, A. (s.d.). Le livre de JAVA premier langage avec 109 exercices corrigés. Éditions EYROLLES.