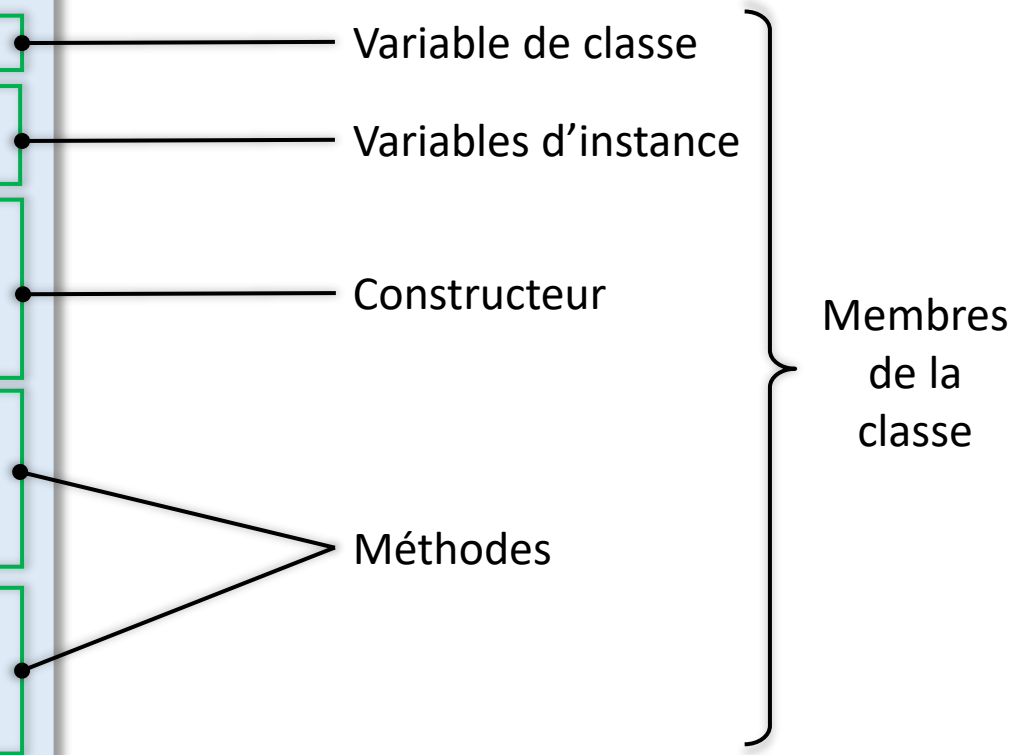
Réutilisation des classes
Délégation
Héritage

dernière modification 16/01/2024 11:38

Classe : rappels

- **Classe**: représente une « famille » d'objets partageant les mêmes propriétés et supportant les mêmes opérations. Elle sert à définir les caractéristiques des objets d'un type donné.
  - *Décrit l'ensemble des données (attributs, caractéristiques, **variables d 'instance**) et des opérations sur données (**méthodes**)*
  - *Sert de « modèle » pour la création d'objets (**instances** de la classe)*

```
public class Point {

    public static final Point ORIGINE = new Point(0,0);

    private int x;  // abscisse du point
    private int y; // ordonnée du point

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

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

    // calcule la distance du point à l'origine
    public double distance() {
      return  Math.sqrt(x * x + y  * y);
    }
 }
```



Réutilisation : introduction

- Comment utiliser une classe comme brique de base pour concevoir d'autres classes ?

- Dans une conception objet on définit des associations (relations) entre classes pour exprimer la réutilisation.

- UML (Unified Modelling Language http://uml.free.fr) définie toute une typologie des associations possibles entre classes. Dans cette introduction nous nous focaliserons sur deux formes d'association

  - *Un objet peut faire appel à un autre objet :* ***délégation***

  - *Un objet peut être créé à partir du « moule » d'un autre objet :* ***héritage***

Délégation

Mise en œuvre

Exemple : la classe Cercle

Agrégation / Composition

Délégation

- Un objet **o1** instance de la classe **Classe1** utilise les services d'un objet **o2** instance de la classe **Classe2** (**o1** délègue une partie de son activité à **o2**)
- La classe **Classe1** utilise les services de la classe **Classe2**
  - *Classe1 est la classe cliente*
  - *Classe2 est la classe serveuse*

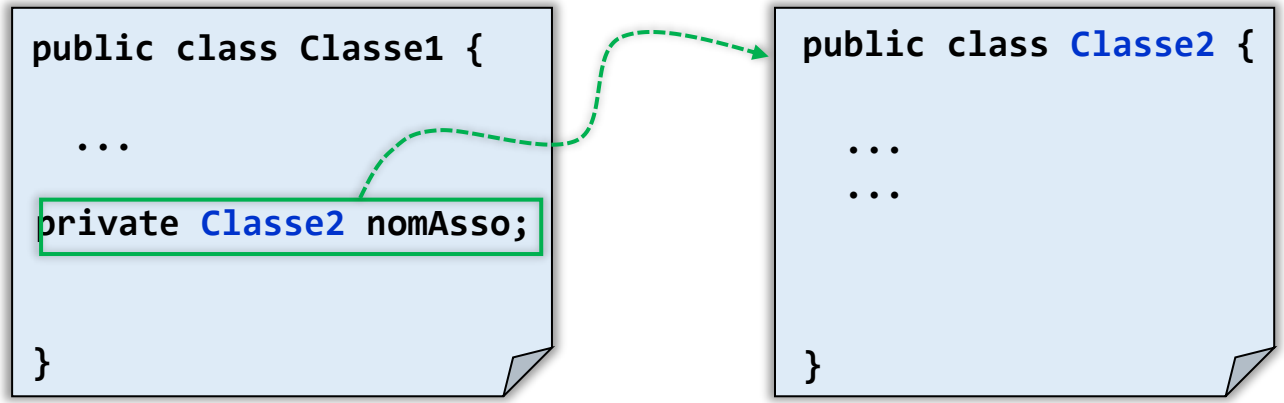
Notation **UML**
(diagramme de classes)

Association à navigabilité restreinte : indique qu'à une instance de **Classe1** peuvent être associées des instances de **Classe2** mais pas l'inverse

1
Cardinalité : indique qu'à une instance de **Classe1** est associée une instance de **Classe2**

nomAsso
permet de nommer l'association

Traduction en java



La classe cliente (**Classe1**) définit une variable d'instance (attribut) de type référence vers une instance (objet) de classe serveuse (**Classe2**)

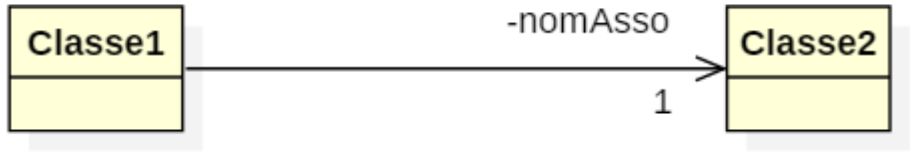
Via la référence **nomAsso** un objet instance de **Classe1** pourra envoyer des messages (déléguer une partie de son activité) à un objet instance de **Classe2**

Délégation

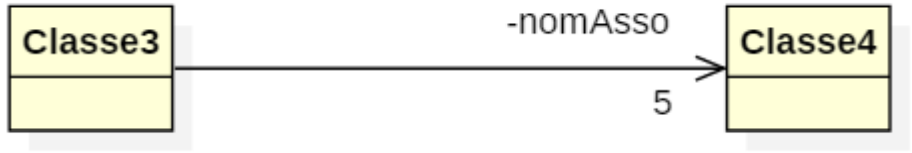
- Associations à navigabilité restreinte et cardinalités

traductions possibles en Java. Il n'y a pas qu'une seule manière d'implémenter ces associations

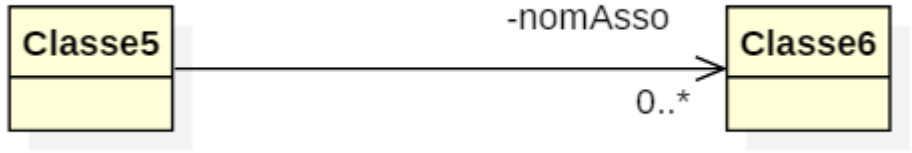
```
public class Classe1 {

    private Classe2 nomAsso;

    ...

}
```




```
public class Classe3 {

    private Classe4[] nomAsso = new Classe4[5];

    ...

}
```

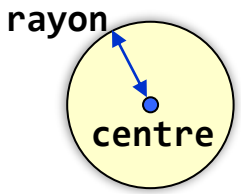



```
import java.util.Set;

public class Classe5 {

    private Set<Classe6> nomAsso;

}
```

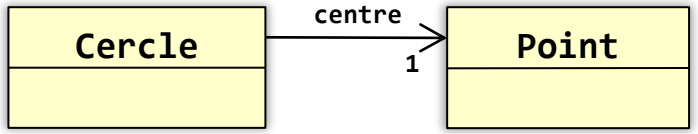


Délégation : exemple

- Exemple la classe **`Cercle`**



- rayon : double
- centre : deux doubles (`x` et `y`) ou bien **`Point`**



L'association entre les classes **`Cercle`** et **`Point`** exprime le fait qu'un cercle **possède** (a un) un centre

```
public class Cercle {

 /**
  * centre du cercle
  */
  private Point centre;

 /**
  * rayon du cercle
  */
  private double r;

  …

  public void translater(double dx, double dy) {
    centre.translater(dx,dy);
  }      le cercle délègue à son centre l'opération de translation
  …
}
```

Délégation : exemple

```
public class Cercle {

 /** centre du cercle */
  private Point centre;

 /** rayon du cercle*/
  private double r;

  public Cercle(Point centre, double r) {


  }

  ...
  public void translater(double dx, double dy) {
    centre.translater(dx,dy);
  }

  public Point getCentre() {

  }

}
```

Délégation : exemple

```
public class Cercle {

 /** centre du cercle */
  private Point centre;

 /** rayon du cercle*/
  private double r;

  public Cercle(Point centre, double r) {
       this.centre = centre;
       this.r = r;
  }

  ...
  public void translater(double dx, double dy) {
    centre.translater(dx,dy);
  }

  public Point getCentre() {
     return this.centre;
  }

}
```

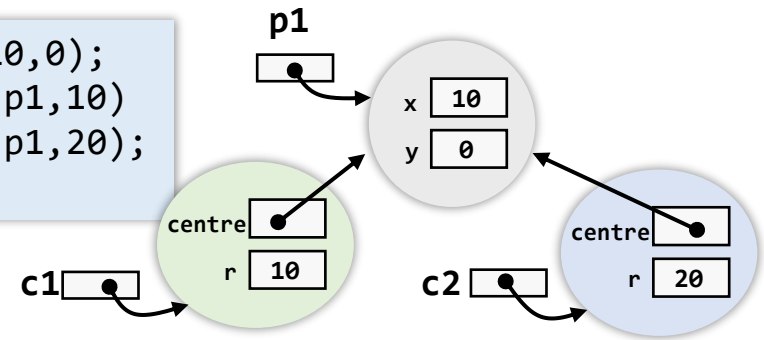
- Le point représentant le centre a une existence autonome
  - *le cercle et son centre ont des cycles de vie indépendants*

Délégation : exemple

```
public class Cercle {

 /** centre du cercle */
  private Point centre;

 /** rayon du cercle*/
  private double r;

  public Cercle(Point centre, double r) {
       this.centre = centre;
       this.r = r;
  }

  ...
  public void translater(double dx, double dy) {
    centre.translater(dx,dy);
  }

  public Point getCentre() {
     return this.centre;
  }

}
```

- Le point représentant le centre a une existence autonome
  - *le cercle et son centre ont des cycles de vie indépendants*
- Ce point peut être partagé
  - *à un même moment il peut être liée à plusieurs instances de cercles (et éventuellement à des instances d'autres classes) .*

```
Point  p1 = new Point(10,0);
Cercle c1 = new Cercle(p1,10)
Cercle c2 = new Cercle(p1,20);
…
```

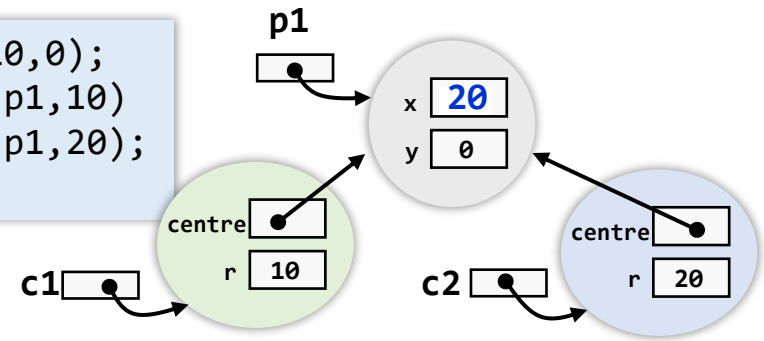


Délégation : exemple

```
public class Cercle {

 /** centre du cercle */
  private Point centre;

 /** rayon du cercle*/
  private double r;

  public Cercle(Point centre, double r) {
       this.centre = centre;
       this.r = r;
  }

  ...
  public void translater(double dx, double dy) {
    centre.translater(dx,dy);
  }

  public Point getCentre() {
     return this.centre;
  }

}
```

- Le point représentant le centre a une existence autonome
  - *le cercle et son centre ont des cycles de vie indépendants*
- Ce point peut être partagé
  - *à un même moment il peut être liée à plusieurs instances de cercles (et éventuellement à des instances d'autres classes) .*

```
Point  p1 = new Point(10,0);
Cercle c1 = new Cercle(p1,10)
Cercle c2 = new Cercle(p1,20);
…
```



- Le point centre d'un cercle peut être utilisé sans passer par le cercle dont il est le centre (attention aux effets de bord)

```
c2.translater(10,0);
```

Affecte aussi cercle **c1**

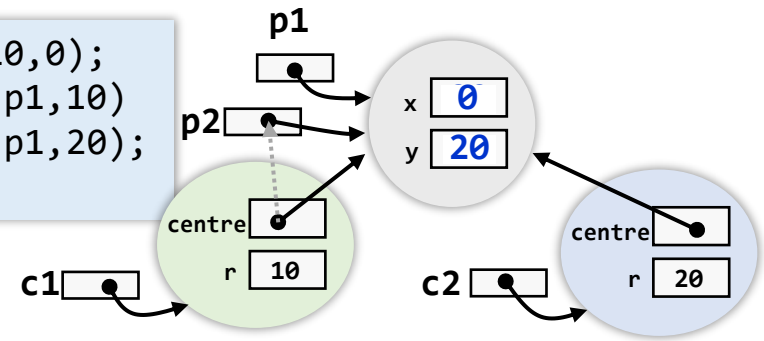
Après ces opérations le centre des 2 cercles est (20,0)

Délégation : exemple

```
public class Cercle {

 /** centre du cercle */
  private Point centre;

 /** rayon du cercle*/
  private double r;

  public Cercle(Point centre, double r) {
       this.centre = centre;
       this.r = r;
  }

  ...
  public void translater(double dx, double dy) {
    centre.translater(dx,dy);
  }

  public Point getCentre() {
     return this.centre;
  }

}
```

- Le point représentant le centre a une existence autonome
  - *le cercle et son centre ont des cycles de vie indépendants*
- Ce point peut être partagé
  - *à un même moment il peut être liée à plusieurs instances de cercles (et éventuellement à des instances d'autres classes) .*

```
Point  p1 = new Point(10,0);
Cercle c1 = new Cercle(p1,10)
Cercle c2 = new Cercle(p1,20);
…
```



- Le point centre d'un cercle peut être utilisé sans passer par le cercle dont il est le centre (attention aux effets de bord)

```
c2.translater(10,0);
```

→ Affecte aussi cercle **c1**
Après ces opérations le centre des 2 cercles est (20,0)

```
p2 = c1.getCentre();
p2.rotation(90);
```

→ Affecte les deux cercles **c1** et **c2**
Après ces opérations le centre des 2 cercles est (0,20)

Délégation : exemple

```
public class Cercle {

 /** centre du cercle */
  private Point centre;

 /** rayon du cercle*/
  private double r;

  public Cercle( Point centre, double r) {
      this.centre = new Point(centre);
      this.r = r;
  }
  …

  public void translater(double dx, double dy) {
    centre.translater(dx,dy);
  }

   public Point getCentre() {
      return this.centre;
   }
}
```

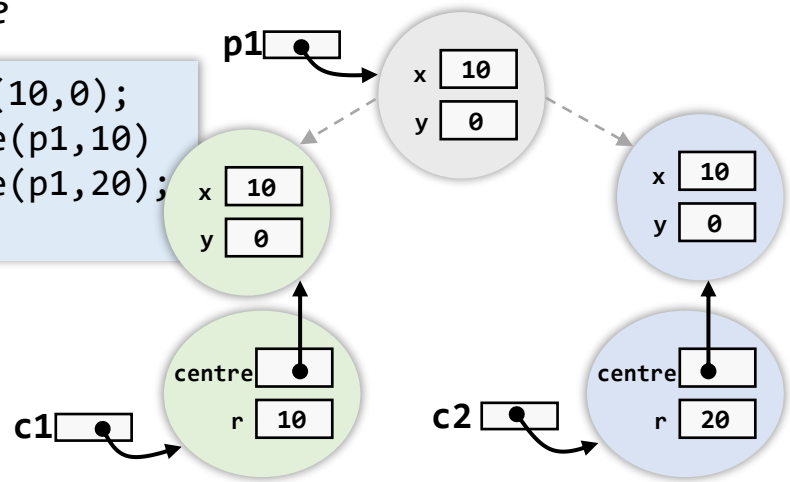
- Le `Point` représentant le centre n'est pas partagé
  - *à un même moment, une instance de* ***`Point`*** *ne peut être liée qu'à un seul* ***`Cercle`***

Délégation : exemple

```
public class Cercle {

 /** centre du cercle */
  private Point centre;

 /** rayon du cercle*/
  private double r;

  public Cercle( Point centre, double r) {
      this.centre = new Point(centre);
      this.r = r;
  }
  …

  public void translater(double dx, double dy) {
    centre.translater(dx,dy);
  }

   public Point getCentre() {
      return this.centre;
   }
}
```

- Le `Point` représentant le centre n'est pas partagé
  - *à un même moment, une instance de* ***`Point`*** *ne peut être liée qu'à un seul* ***`Cercle`***

```
Point  p1 = new Point(10,0);
Cercle c1 = new Cercle(p1,10)
Cercle c2 = new Cercle(p1,20);
...
```

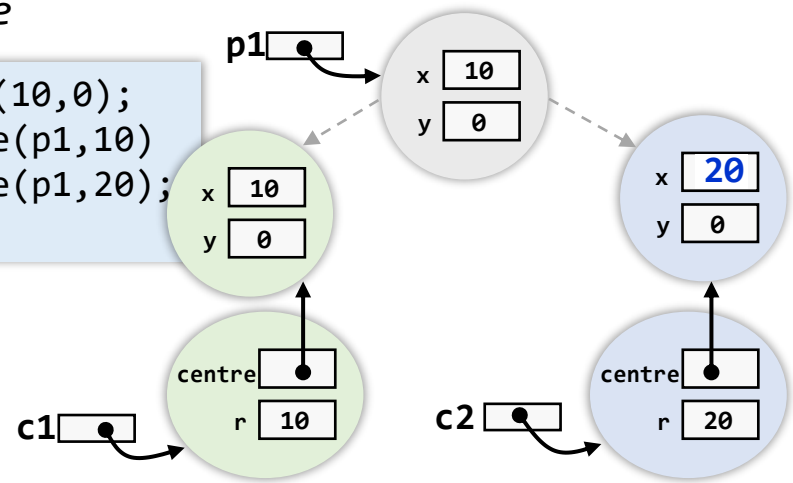


Délégation : exemple

```
public class Cercle {

 /** centre du cercle */
  private Point centre;

 /** rayon du cercle*/
  private double r;

  public Cercle( Point centre, double r) {
      this.centre = new Point(centre);
      this.r = r;
  }
  …

  public void translater(double dx, double dy) {
    centre.translater(dx,dy);
  }

   public Point getCentre() {
      return this.centre;
   }
}
```

- Le **Point** représentant le centre n'est pas partagé
  - *à un même moment, une instance de **Point** ne peut être liée qu'à un seul **Cercle***

```
Point  p1 = new Point(10,0);
Cercle c1 = new Cercle(p1,10)
Cercle c2 = new Cercle(p1,20);
...
```



- Le point centre d'un cercle ne peut plus être utilisé sans passer par le cercle dont il est le centre (plus d'effets de bord)

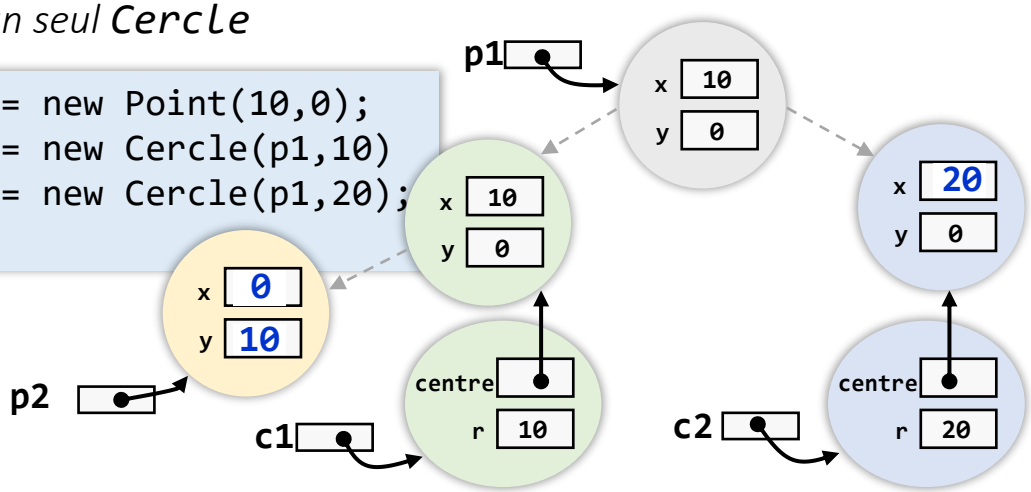
```
c2.translater(10,0);
```
→ N'affecte que le cercle **c2**

Délégation : exemple

```
public class Cercle {

 /** centre du cercle */
  private Point centre;

 /** rayon du cercle*/
  private double r;

  public Cercle( Point centre, double r) {
      this.centre = new Point(centre);
      this.r = r;
  }
  …

  public void translater(double dx, double dy) {
    centre.translater(dx,dy);
  }

   public Point getCentre() {
      return new Point(this.centre);
   }
}
```

- Le **Point** représentant le centre n'est pas partagé
  - *à un même moment, une instance de **Point** ne peut être liée qu'à un seul **Cercle***

```
Point  p1 = new Point(10,0);
Cercle c1 = new Cercle(p1,10)
Cercle c2 = new Cercle(p1,20);
...
```



- Le point centre d'un cercle ne peut plus être utilisé sans passer par le cercle dont il est le centre (plus d'effets de bord)

```
c2.translater(10,0);
```
→ N'affecte que le cercle **c2**

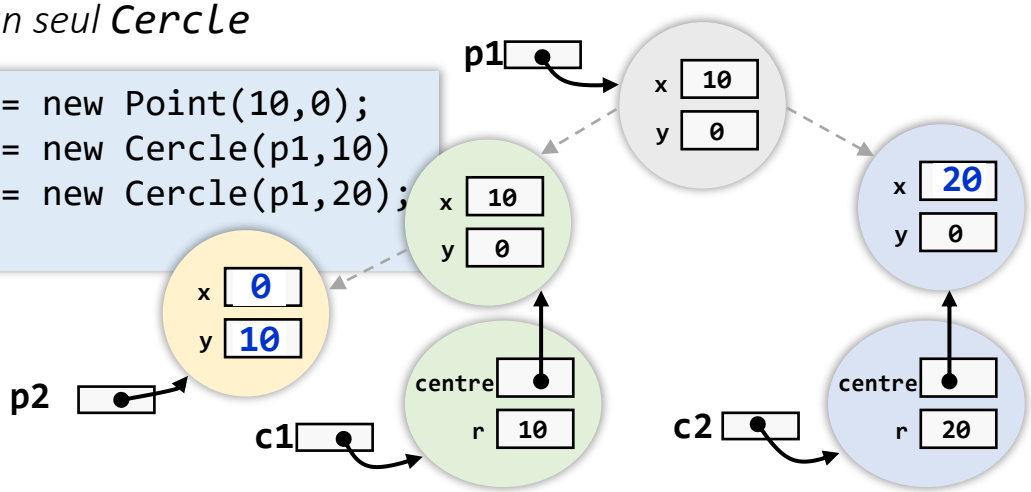
```
p2 = c1.getCentre();
p2.rotation(90);
```
→ affecte le cercle **c1** mais pas **c2**

Délégation : exemple

```
public class Cercle {

 /** centre du cercle */
  private Point centre;

 /** rayon du cercle*/
  private double r;

  public Cercle( Point centre, double r) {
      this.centre = new Point(centre);
      this.r = r;
  }
  …

  public void translater(double dx, double dy) {
    centre.translater(dx,dy);
  }

   public Point getCentre() {
      return new Point(this.centre);
   }
}
```

- Le **Point** représentant le centre n'est pas partagé
  - *à un même moment, une instance de **Point** ne peut être liée qu'à un seul **Cercle***

```
Point  p1 = new Point(10,0);
Cercle c1 = new Cercle(p1,10)
Cercle c2 = new Cercle(p1,20);
...
```



- Le point centre d'un cercle ne peut plus être utilisé sans passer par le cercle dont il est le centre (plus d'effets de bord)

```
c2.translater(10,0);
```
→ N'affecte que le cercle **c2**

```
p2 = c1.getCentre();
p2.rotation(90);
```
→ N'affecte pas les cercles **c1** et **c2**

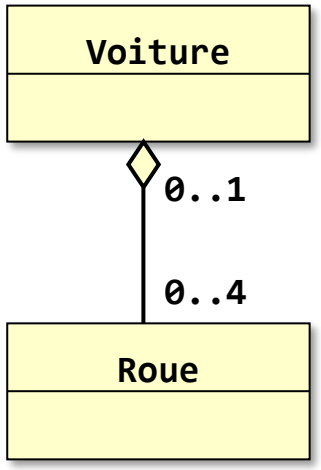
- les cycles de vies du **Point** et du **Cercle** sont liés : si le cercle est détruit (ou copié), le centre l'est aussi.

Agrégation / Composition

- Les deux exemples précédents traduisent deux nuances (sémantiques) de l'association a-un entre la classe `Cercle` et la classe `Point`

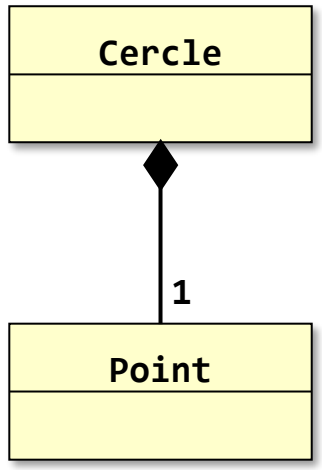
- UML distingue ces deux sémantiques en définissant deux type de relations :

Agrégation



L'élément agrégé (`Roue`) a une existence autonome en dehors de l'agrégat (`Voiture`)

Composition (Agrégation forte)



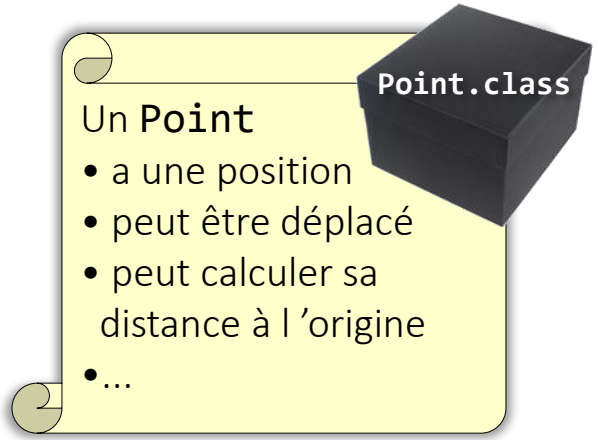
A un même moment, une instance de composant (`Point`) ne peut être liée qu'à un seul agrégat (`Cercle`), et le composant a un cycle de vie dépendant de l'agrégat.

Héritage

- Dérivation (extension) d'une classe
- Terminologie
- Généralisation/spécialisation
- Héritage en Java
- Redéfinition des méthodes
- Réutilisation
- Chaînage des constructeurs
- Visibilité des variables et des méthodes
- Classes et méthodes finales
- Classes scellées

Exemple introductif

- Le problème
  - *une application a besoin de services dont une partie seulement est proposée par une classe déjà définie (classe dont on ne possède pas nécessairement le source)*
  - *ne pas réécrire le code*



Application a besoin
• de manipuler des points (comme le permet la classe `Point`)
• mais en plus de les dessiner sur l'écran.

`PointGraphique = Point`
+ une couleur
+ une opération d'affichage

- Solution en POO : l'**héritage** (***inheritence***)
  - *définir une nouvelle classe à partir de la classe déjà existante*

Héritage

- La classe **PointGraphique** hérite (dérive) de la classe **Point**

Point.java

```
public class Point {
   double x;
   double y;

   void translater(double dx, double dy) {
       x += dx;
       y += dy;
   }

   double distance() {
       return Math.sqrt(x*x + y*y);
   }
}
```

PointGraphique.java

```
import java.awt.Color;
import java.awt.Graphics;

public class PointGraphique extends Point {

    Color coul;

    // constructeur
    public PointGraphique(double x, double y, Color c) {
       this.x = x;
       this.y = y;
       this.coul = c;
    }

    // affiche le point matérialisé par
    // un rectangle de 3 pixels de coté
    public void dessine(Graphics g) {
       g.setColor(coul);
       g.fillRect((int) x - 1,(int)y - 1,3,3);
    }
}
```

PointGraphique hérite de (étend) Point un PointGraphique possède les variables et méthodes définies dans la classe **Point**

PointGraphique définit un nouvel attribut

Attributs hérités de la classe **Point**

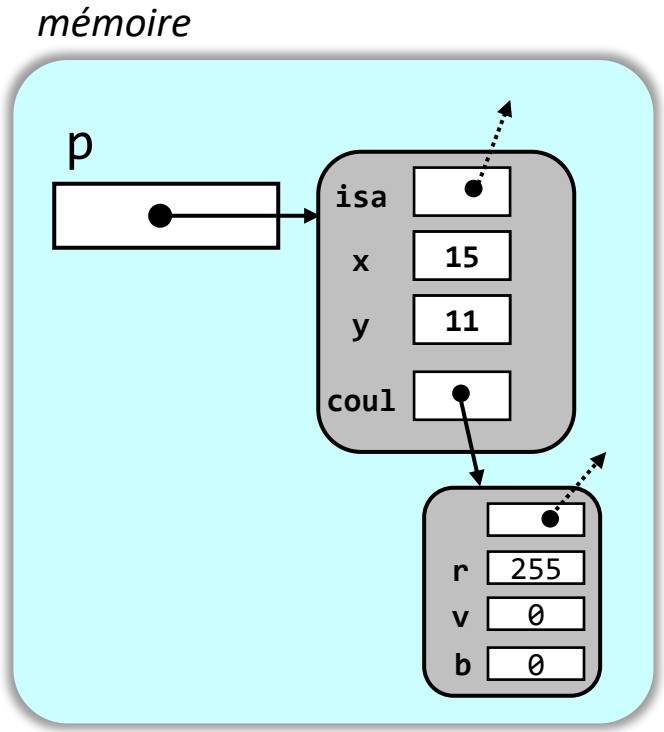
PointGraphique définit une nouvelle méthode

Utilisation des instances d'une classe héritée

- Un objet instance de PointGraphique possède les attributs définis dans **PointGraphique** ainsi que les attributs définis dans **Point** (un **PointGraphique** est aussi un **Point**)

- Un objet instance de **PointGraphique** répond aux messages définis par les méthodes décrites dans la classe **PointGraphique** **et aussi** à ceux définis par les méthodes de la classe **Point**

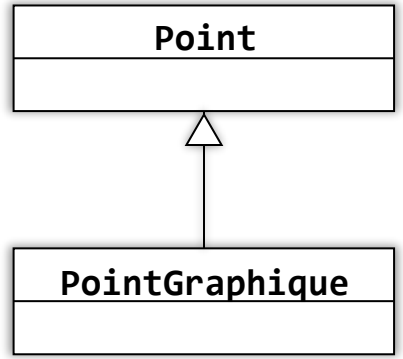
```
PointGraphique p = new PointGraphique();

// utilisation des variables d'instance héritées
p.x = 15;
p.y = 11;

// utilisation d'une variable d'instance spécifique
p.coul = new Color(255,0,0);

// utilisation d'une méthode héritée
double dist = p.distance();

// utilisation d'une méthode spécifique
p.dessine(graphicContext);
```



- Héritage permet de reprendre les caractéristiques d'une classe `M` existante pour les étendre et définir ainsi une nouvelle classe `F` qui hérite de `M`.

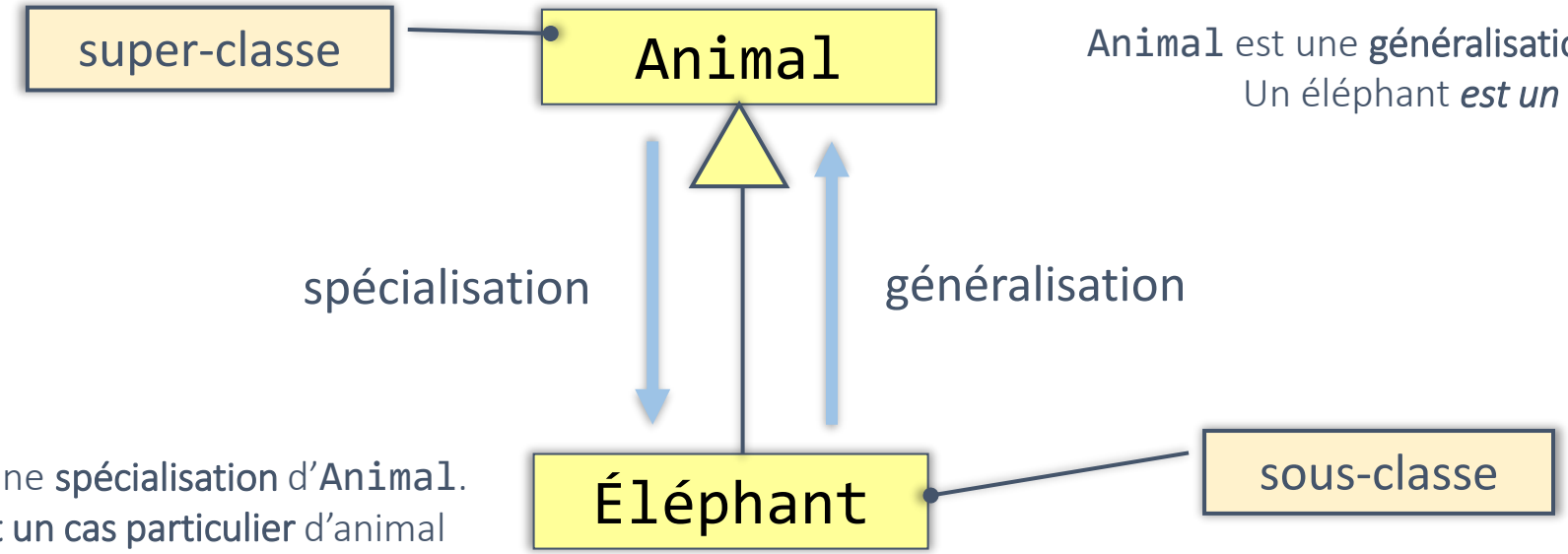
- Les objets instance de `F` possèdent toutes les caractéristiques des instances de `M` avec en en plus celles définies dans `F`

  - *`Point` est la classe mère et `PointGraphique` la classe fille.*
  - *la classe `PointGraphique` hérite de la classe `Point`*
  - *la classe `PointGraphique` est une classe dérivée de la classe `Point`*
  - *la classe `PointGraphique` est une sous-classe de la classe `Point`*
  - *la classe `Point` est la super-classe de la classe `PointGraphique`*



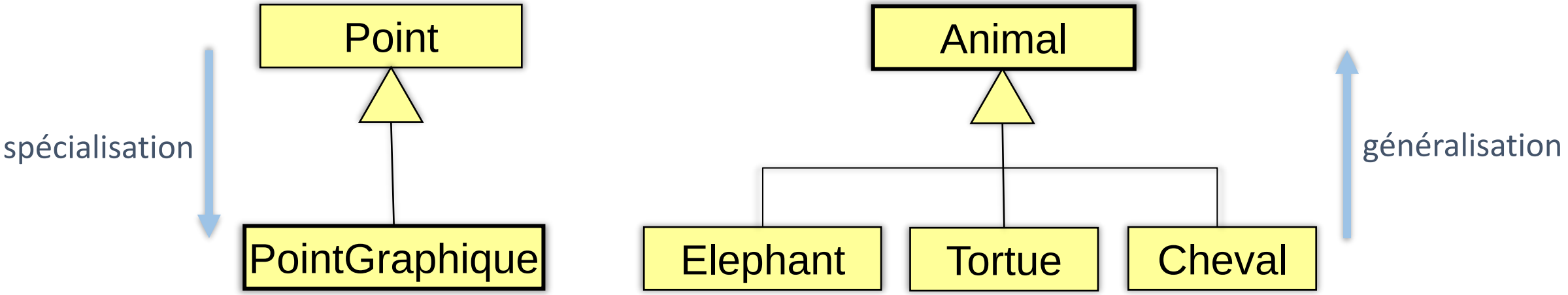
- la relation d'héritage peut être vue comme une relation de "généralisation/spécialisation" entre une classe (la *super-classe*) et plusieurs classes plus spécialisées (ses *sous-classes*).

- La généralisation exprime une relation “**est-un**” entre une classe et sa super-classe (chaque instance de la classe est aussi décrite de façon plus générale par la super-classe).



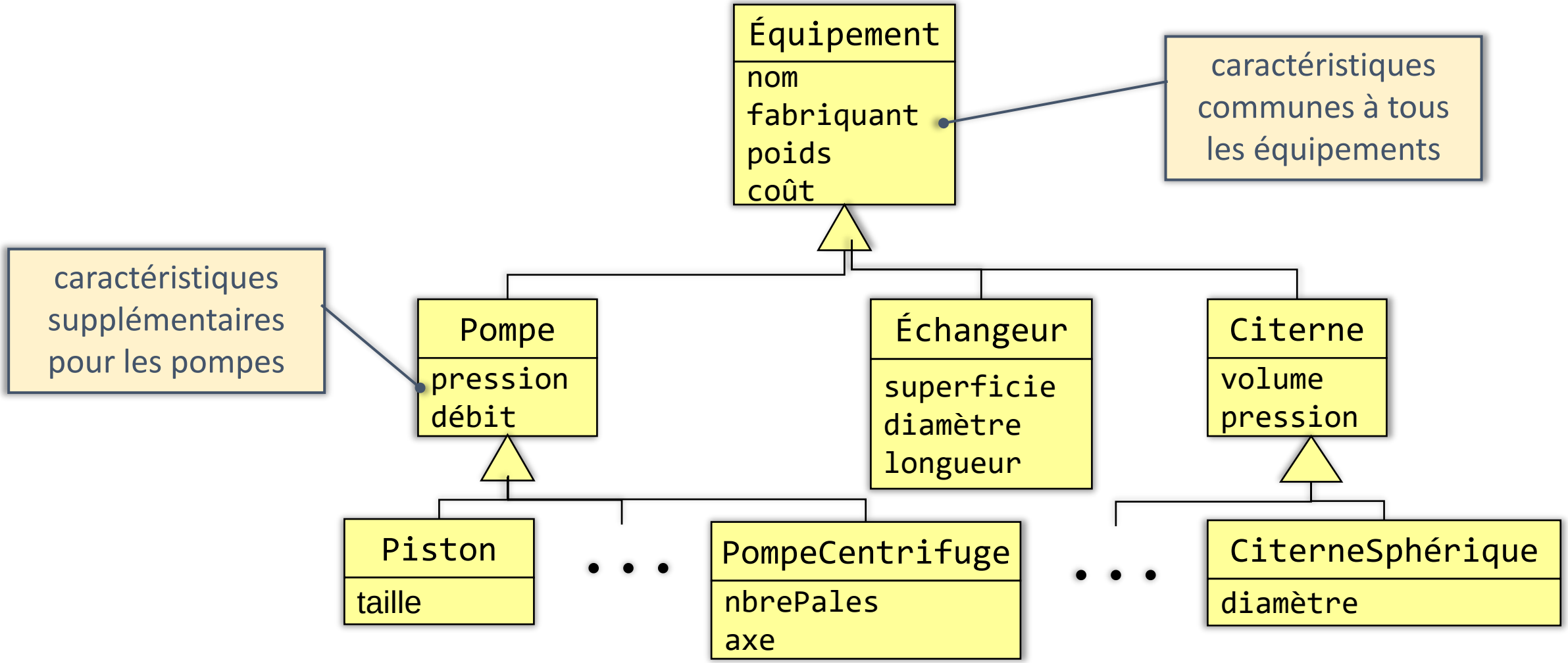
- La spécialisation exprime une relation de “**particularisation**” entre une classe et sa sous-classe (chaque instance de la sous-classe est décrite de manière plus spécifique)

- Utilisation de l'héritage :
  - *dans le sens "spécialisation" pour **réutiliser** par modification incrémentielle les descriptions existantes.*
  - *dans le sens "généralisation" pour **abstraire** en factorisant les propriétés communes aux sous-classes,*



Héritage

- pas de limitation dans le nombre de niveaux dans la hiérarchie d'héritage
- méthodes et variables sont héritées au travers de tous les niveaux



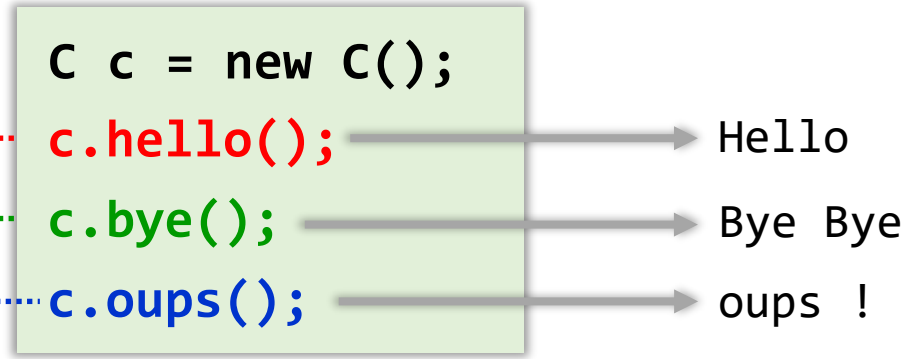
Héritage à travers tous les niveaux

```
public class A {

  public void hello() {
    System.out.println("Hello");
  }
}
```

```
public class B extends A {

  public void bye() {
   System.out.println("Bye Bye");
  }
}
```

```
public class C extends B {

  public void oups() {
    System.out.println("oups!");
  }
}
```

- Pour résoudre un message, la hiérarchie des classes est parcourue de manière ascendante jusqu'à trouver la méthode correspondante.



Redéfinition des méthodes

- une sous-classe peut **ajouter** des variables et/ou des méthodes à celles qu'elle hérite de sa super-classe.
- une sous-classe peut **redéfinir** (*override*) les méthodes dont elle hérite et fournir ainsi des implémentations spécialisées pour celles-ci.
- **Redéfinition d'une méthode (method overriding)**
  - *lorsque la classe définit une méthode dont le nom, le type de retour et le type des arguments sont identiques à ceux d'une méthode dont elle hérite*
- Lorsqu'une méthode redéfinie par une classe est invoquée pour un objet instance de cette classe, c'est la nouvelle définition et non pas celle de la super-classe qui est invoquée.

Redéfinition des méthodes

```
public class A {

  public void affiche() {
    System.out.println("Je suis un A");
  }

  public void hello() {
    System.out.println("Hello");
  }
}
```

```
public class B extends A {

  public void affiche() {
    System.out.println("Je suis un B");
  }
}
```

```
A a = new A();
B b = new B();

a.affiche();   → Je suis un A
a.hello();     → Hello

b.hello();     → Hello
b.affiche();   → Je suis un B
```

la méthode `affiche()` est redéfinie
c'est la méthode la plus spécifique qui est exécutée

Redéfinition des méthodes

- Ne pas confondre **redéfinition** (*overriding*) avec **surcharge** (*overloading*)

```
public class A {

  public void methodX(int i) {
         ...
  }
}
```

Surcharge

```
public class B extends A {

  public void methodX(Color i) {
         ...
  }
}
```

Redéfinition

```
public class C extends A {

  public void methodX(int i) {
         ...
  }
}
```

`B` possède deux méthodes `methodX`
`methodX(int)` et `methodX(Color)`

`C` possède une seule méthode `methodX`
`methodX(int)`

Redéfinition des méthodes

- **Annotations[1] (java 1.5+)** : métadonnées sur un programme (données qui ne font pas partie du programme lui-même)
  - *informations pour le compilateur (détection d'erreurs)*
  - *traitements à la compilation ou au déploiement (génération de code, de fichiers XML, …)*
  - *traitement à l'exécution*
- Lors d'une redéfinition utiliser l'annotation `@Override`
- Evite de faire une surcharge alors que l'on veut faire une redéfinition

ClasseA.java

```java
public class ClasseA {

    protected double x;

    public void add(double x) {
        System.out.println("add de ClasseA");
        this.x +=x;
    }
}
```

ClasseB.java

```java
public class ClasseB extends ClasseA {

    @Override
    public void add(double x) {
        System.out.println("add de ClasseA");
        this.x += x;
    }

}
```

indique au compilateur que l'on veut faire une redéfinition de méthode

ClasseC.java 1

```java
public class ClasseC extends ClasseA {

    @Override
    public void add(int x) {
```

ClasseC.java 1 of 1 problem

The method add(int) of type ClasseC must override or implement a supertype method Java(67109498)

```java
        System.out.println("add de ClasseC");
        this.x +=x;
    }

}
```

[1] Pour en savoir plus sur les annotations : http://adiguba.developpez.com/tutoriels/java/tiger/annotations/

Redéfinition des méthodes

- Redéfinition d'une méthode (method overriding)
  - lorsque la classe définit une méthode dont le nom, le type de retour et le type des arguments sont identiques à ceux d'une méthode dont elle hérite
  - avant Java 5 : le type de retour doit être le même
  - Java 5 et + : le type du résultat peut être une sous-classe

```
public class A { … }
public class B extends A { … }
```

```
public class C1 {

  public A uneMethode() { … }

}
```

Le type du résultat de `uneMethode` est `A`

```
public class C2 extends C1 {
  @Override
  public B uneMethode() { … }

}
```

Le type du résultat de `uneMethode` est `B`

`B` est une sous classe de `A`

`B uneMethode()` dans `C2` rédéfinit `A uneMethode()` de `C1`

Redéfinition avec réutilisation

- **Redéfinition** des méthodes (method **overriding**) :
  - possibilité de réutiliser le code de la méthode héritée (**super**)

```
public class Etudiant {
   String nom;
   String prénom;
   int age;
   ...
   public void affiche() {
     System.out.println("Nom : " + nom + " Prénom : " + prénom);
     System.out.println("Age : " + age);
     ...
   }
   ...
 }
```

```
public class EtudiantSportif extends Etudiant {
   String sportPratiqué;
   ...
   public void affiche(){
     super.affiche();

     System.out.println("Sport pratiqué : "+ sportPratiqué);
     ...
   }
 }
```

this permet de faire référence à l'objet récepteur du message
super permet de désigner la superclasse

Un appel **super.*uneméthode(...)*** peut être effectué n'importe où dans le corps de la méthode (à ne pas confondre avec **super(…)** qui permet d'invoquer un autre constructeur de la classe et qui doit nécessairement être la première instruction du constructeur où il est invoqué voir slide 38).

Particularités de l'héritage en Java

- Héritage simple
  - une classe ne peut hériter que d'une seule autre classe
    - dans certains autres langages (ex C++) possibilité d'héritage multiple
- Héritage multiple
  - une classe peut hériter de plus d'une seule classe
  - problèmes de l'héritage multiple : ex héritage en diamant

Diamond inheritance

- combien d'écrans possède un smartphone ?
- si plusieurs implémentations de la méthode charger laquelle est invoquée ?

Les langages de programmation peuvent résoudre ce problème de façons différentes :

- Une solution consiste à rajouter un mécanisme dans le langage pour choisir entre la fusion des entités répétées ou le renommage de celles-ci afin de séparer les entités (Eiffel, Ocaml).
- Héritage virtuel dans C++
- …

Mécanismes complexes pas toujours très bien compris et utilisés par les programmeurs.
➔ certains langages on fait le choix de ne pas proposer l'héritage multiple (Java, C#, Ruby, ObjectiveC)

- Héritage simple
  - une classe ne peut hériter que d'une seule autre classe

- La hiérarchie d'héritage **est un arbre** dont la racine est la classe `Object` (`java.lang`)
  - toute classe autre que `Object` possède une super-classe
  - toute classe hérite directement ou indirectement de la classe `Object`
  - par défaut une classe qui ne définit pas de clause `extends` hérite de la classe `Object`

```
public class Point extends Object {

    int x;  // abscisse du point
    int y; // ordonnée du point

    ...
 }
```

- Principales méthodes de la classe **Object**

- `public final Class getClass()`
  Renvoie la référence de l'objet Java représentant la classe de l'objet

- `public boolean equals(Object obj)`
  Teste l'égalité de l'objet avec l'objet passé en paramètre

```
return (this == obj);
```

(on en reparlera lors du cours sur le polymorphisme)

- `protected Object clone()`
  Crée un copie de l'objet

- `public int hashCode()`
  Renvoie une clé de hashcode pour adressage dispersé

(on en reparlera lors du cours sur les collections)

- `public String toString()`
  Renvoie une chaîne représentant la valeur de l'objet

```
return getClass().getName() + "@"
   + Integer.toHexString(hashCode());
```

A propos de toString

Opérateur de concaténation

<Expression de type String> **+** *<reference>*

<=>

<Expression de type String> **+** *<reference>*.**toString()**

du fait que la méthode **toString** est définie dans la classe **Object**, le compilateur Java est sûr que quel que soit le type (la classe) de l'objet il saura répondre au message **toString()**

```
public String toString(){
  return getClass().getName() + "@" +
      Integer.toHexString(hashCode());
}
```

```
public class Object {
  ...
}
```

```
public class Point {
  private double x;
  private double y;

  ...
}
```

```
@0verride
public String toString(){
   return "Point:[" + x + "," + y + "]");
}
```

La classe **Point** ne redéfinit pas **toString**

```
Point@2a340e
```

```
Point p = new Point(15,11);
System.out.println(p);
```

La classe **Point** redéfinit **toString**

```
Point:[15.0,11.0]
```

Réutilisation des constructeurs

- Redéfinition des méthodes (method overriding) :
  - possibilité de réutiliser le code de la méthode héritée (`super`)

- De la même manière il est important de pouvoir réutiliser le code des constructeurs de la super classe dans la définition des constructeurs d'une nouvelle classe
  - invocation d'un constructeur de la super classe :

```
super(paramètres du constructeur)
```

  - utilisation de `super(…)` analogue à celle de `this(…)`
    - ne peut être faite que dans un constructeur
    - doit être la première instruction du constructeur

Réutilisation des constructeurs

```
public class Point {
    double x,y;

    public Point(double x, double y) {
        this.x = x; this.y = y;
    }
    ...
}
```

```
public class PointCouleur extends Point {

    Color c;

    public PointCouleur(double x, double y, Color c) {
        super(x,y);
        this.c = c;
    }
    ...
}
```

Appel du constructeur de la super-classe.
Cet appel, si il est présent, **doit toujours** être la première instruction du corps du constructeur.

Chaînage des constructeurs

- appel à un constructeur de la super classe doit **toujours** être la première instruction dans le corps du constructeur
  - si la première instruction d'un constructeur n'est pas un appel explicite à l'un des constructeur de la superclasse, alors JAVA insère implicitement l'appel `super()`
  - chaque fois qu'un objet est créé les constructeurs sont invoqués en remontant en séquence de classe en classe dans la hiérarchie jusqu'à la classe `Object`
  - c'est le corps du constructeur de la classe `Object` qui est toujours exécuté en premier, suivi du corps des constructeurs des différentes classes en redescendant dans la hiérarchie.

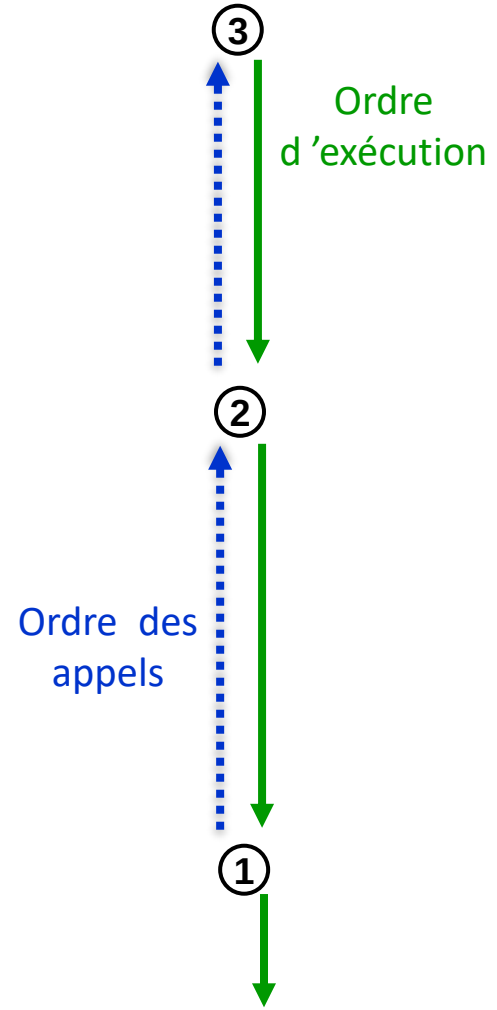
- garantit qu'un constructeur d'une classe est toujours appelé lorsqu'une instance de l'une de ses sous classes est créée
  - un objet `c`, instance de `ClasseC` sous classe de `ClasseB` elle même sous classe de `ClasseA` est un objet de classe `ClasseC` mais est aussi un objet de classe `ClasseB` et un objet de classe `ClasseA`.
    Lorsqu'il est créé `c`, doit l'être avec les caractéristiques d'un objet de `ClasseA` de `ClasseB` et de `ClasseC`

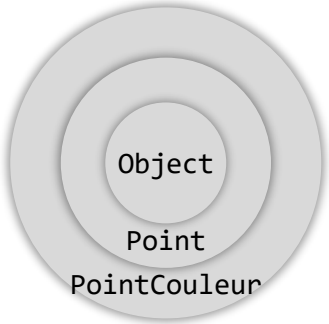
Chaînage des constructeurs

```
public class Object {

    public Object()  ③
    {
          ...
    }
    ...
}
```

```
public class Point  extends Object  {
    double x,y;

    public Point(double x, double y) ②
    {
          super();  // appel implicite
          this.x = x; this.y = y;
    }
    ...
}
```

```
public class PointCouleur extends Point {

    Color c;

    public PointCouleur(double x, double y, Color c) ①
    {
          super(x,y);
          this.c = c;
    }
    ...
}
```





Constructeur par défaut

- Lorsqu'une classe ne définit pas explicitement de constructeur, elle possède un constructeur par défaut :
  - sans paramètres
  - de corps vide
  - inexistant si un autre constructeur existe

```
public class Object {

    public Object()
    {
            ...
    }
    ...
}
```

```
public class A extends Object {

    // attributs
    String nom;

    // méthodes
    String getNom() {
      return nom;
    }
    ...
}
```

```
public A() {
  super();
}
```

Constructeur par défaut implicite

Garantit chaînage des constructeurs

Constructeur par défaut

```
public class ClasseA {
    double x;
    // constructeur
    public ClasseA(double x) {
           this.x = x;
    }
}
```

Constructeur explicite
masque constructeur par défaut

Pas de constructeur sans paramètres dans `ClasseA`

```
public class ClasseB extends ClasseA {
    double y = 0;

    // pas de constructeur
}
```

```
public ClasseB(){
  super();
}
```

Constructeur par défaut implicite

```
C:>javac ClasseB.java
ClasseB.java:3: No constructor matching ClasseA() found
in class ClasseA.
   public ClasseB() {
          ^
1 error

Compilation exited abnormally with code 1 at Sat Jan 29
09:34:24
```

Redéfinition des attributs (Shadowed variables)

- Lorsqu'une sous classe définit une variable d'instance dont le nom est identique à l'une des variables dont elle hérite, **la nouvelle définition masque la définition héritée**
  - *l'accès à la variable héritée se fait au travers de* **`super`**

```
public class ClasseA {
    int x;

    ...
}
```

```
public class ClasseB extends Classe A {
    double x;

    ...
}
```

```
public class ClasseC extends Classe B {
    char x;

    ...
}
```

Dans le code de `ClasseC`

```
((ClasseA)this).x
```

~~`super.super.x`~~

```
super.x
```

```
x ou this.x
```

en général ce n'est pas
une très bonne idée
de masquer les variables

Visibilité des variables et méthodes

- principe **d'encapsulation** : les données propres à un objet ne sont accessibles qu'au travers des méthodes de cet objet
  - sécurité des données : elles ne sont accessibles qu'au travers de méthodes en lesquelles on peut avoir confiance
  - masquer l'implémentation : l'implémentation d'une classe peut être modifiée sans remettre en cause le code utilisant celle-ci

- en JAVA possibilité de contrôler l'accessibilité (visibilité) des membres (variables et méthodes) d'une classe
  - `public` membre accessible à toute autre classe
  - `private` le membre n'est accessible qu'à l'intérieur de la classe où il est défini
  - `protected` le membre est accessible
    - dans la classe où il est défini,
    - dans toutes ses sous-classes
    - et dans toutes les classes du même package
  - – (visibilité par défaut `package`) le membre n'est accessible que dans les classes du même package que celui de la classe où il est défini

Visibilité des variables et méthodes

	private	- (package)	protected	public
La classe elle même	oui	oui	oui	oui
Classes du même package	non	oui	oui	oui
Sous-classes d'un autre package	non	non	oui	oui
Classes (non sous-classes) d'un autre package	non	non	non	oui

Héritage d'attributs privés

Point.java

```
public class Point {
  private double x;
  private double y

  ...

  public double getX() {
    return x;
  }

  public double getY() {
     return y;
  }
  ...
}
```

Les attributs sont privés dans la super-classe on ne peut les utiliser directement dans le code de la sous-classe

PointGraphique.java

```
import java.awt.Color;
import java.awt.Graphics;
public class PointGraphique extends Point {

   Color coul;

   // constructeur
   public void PointGraphique(double x, double y,
                    Color c) {
      super(x,y);
      this.coul = c;
   }

   // affiche le point matérialisé par
   // un rectangle de 3 pixels de coté
   public void dessine(Graphics g) {
      g.setColor(coul);
      g.fillRect((int) x - 1,(int) y - 1,3,3);
   }
}
```

getX() **gety()**

Attributs hérités de la classe **Point**

Héritage d'attributs protégés

Point.java

```
public class Point {
  protected double x;
  protected double y

  ...

  public double getX() {
    return x;
  }

  public double getY() {
     return y;
  }
  ...
}
```

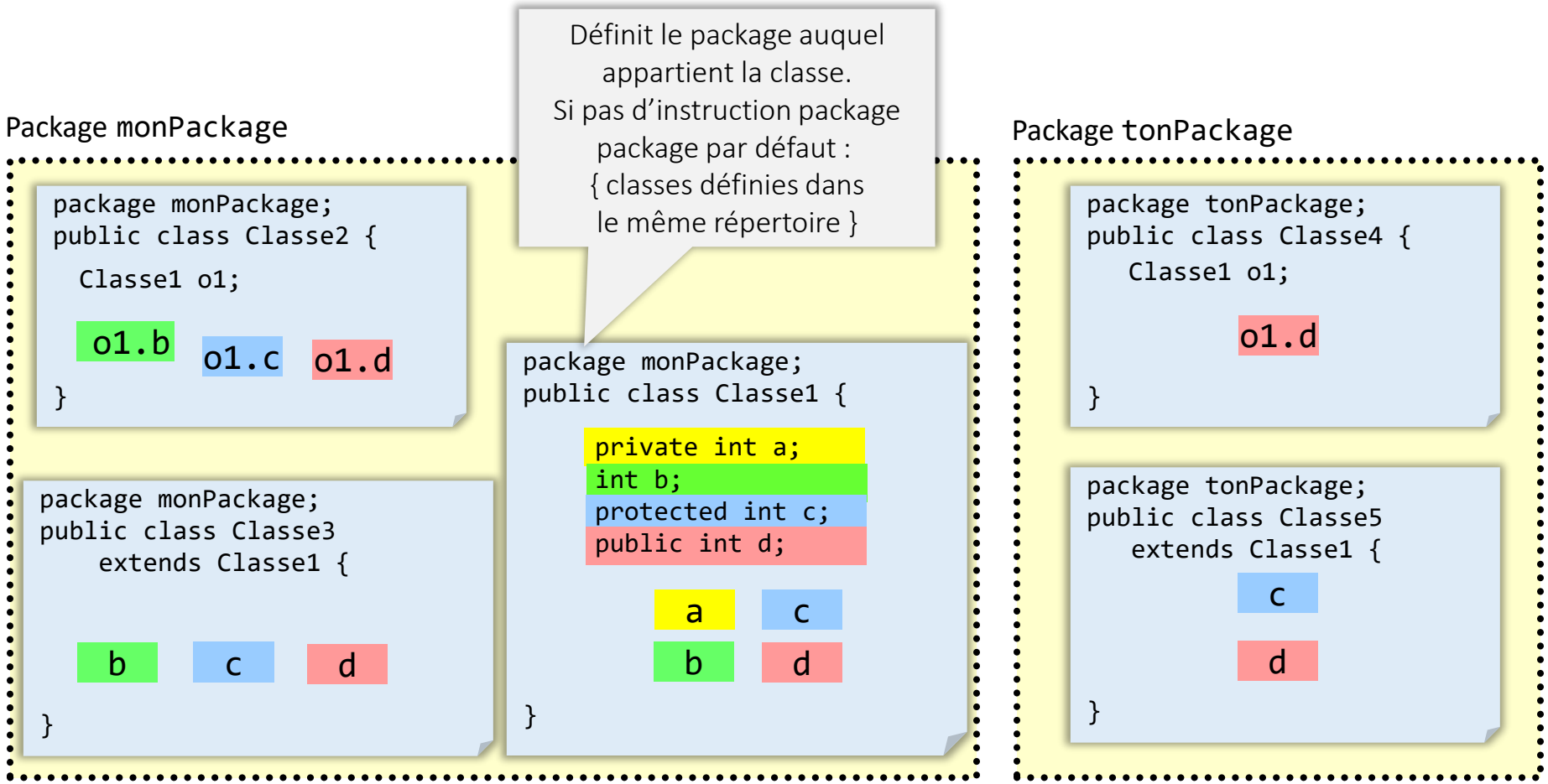
Les attributs sont protégés dans la super-classe on peut les utiliser directement dans le code de la sous-classe

PointGraphique.java

```
import java.awt.Color;
import java.awt.Graphics;
public class PointGraphique extends Point {

   Color coul;

   // constructeur
   public void PointGraphique(double x, double y,
                    Color c) {
      super(x,y);
      this.coul = c;
   }

   // affiche le point matérialisé par
   // un rectangle de 3 pixels de coté
   public void dessine(Graphics g) {
      g.setColor(coul);
      g.fillRect((int) x - 1,(int) y - 1,3,3);
   }
}
```

Attributs hérités de la classe **Point**

Visibilité des variables et méthodes



Les mêmes règles de visibilité s'appliquent aux méthodes

Visibilité des classes

- Deux niveaux de visibilité pour les classes :
  - `public` : la classe peut être utilisée par n'importe quelle autre classe
  - - (`package`) : la classe ne peut être utilisée que par les classes appartenant au même package

`ClassB` définie dans le même package, pas besoin d'instruction import

`ClassA` définie dans le même package, pas besoin d'instruction import

Package **A**

```
package packageA;

public class ClasseA {

    ClasseB b;
     ...
}
```

```
package packageA;

class ClasseB extends ClasseA {
    ClasseA a;
    ...
}
```

Package **B**

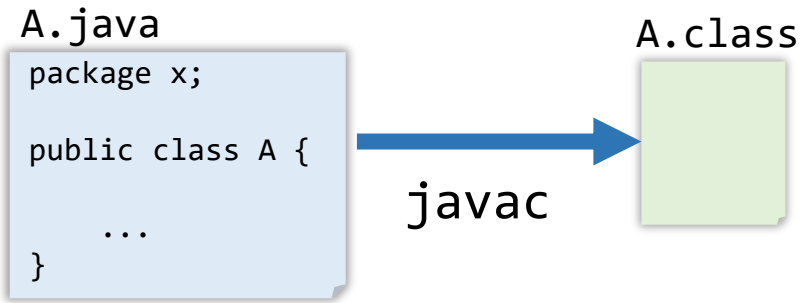
```
package packageB;

import packageA.ClasseA;

public class ClasseC {
    ClasseA a;
    ClasseB b;

}
```

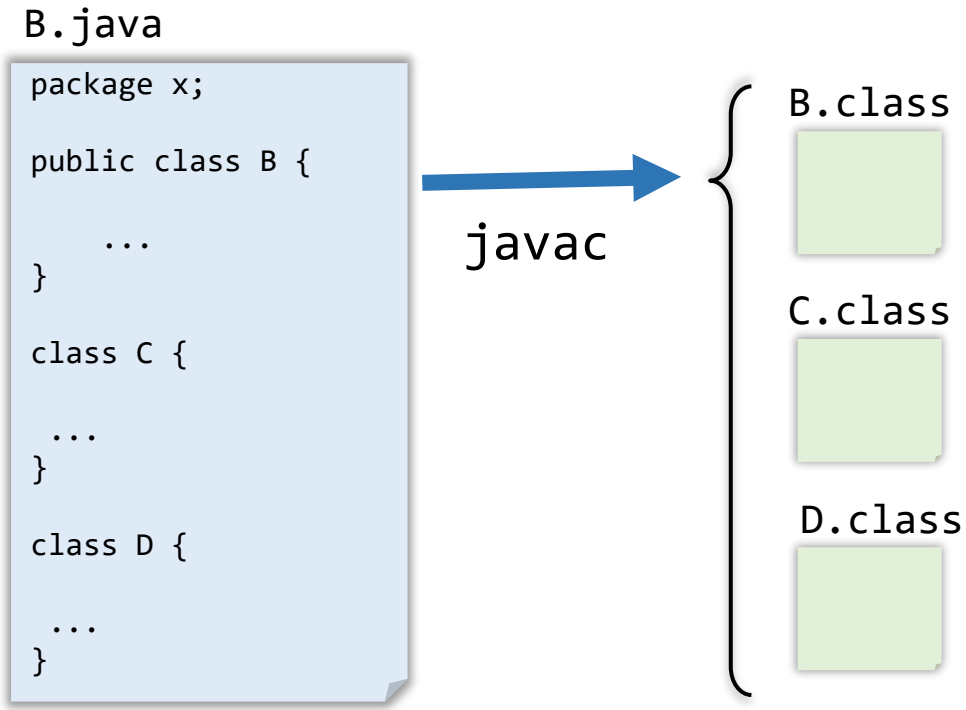
`ClassA` définie dans un autre package, nécessité d'instruction import

`ClassB` définie dans un autre package mais n'est pas publique : impossibilité de l'utiliser

Visibilité des classes

- Jusqu'à présent on a toujours dit :
  - une classe par fichier
  - le nom du fichier source : le nom de la classe avec extension **`.java`**
- En fait la vraie règle est :
  - une classe **publique** par fichier
  - le nom du fichier source : le nom de la classe publique





Méthodes finales

```
public final void méthodeX(...) {

    ...

}
```

- `final` permet de « verrouiller » la méthode pour interdire toute éventuelle redéfinition dans les sous-classes
- efficacité
  - quand le compilateur rencontre un appel à une méthode finale il **peut** remplacer l'appel habituel de méthode (empiler les arguments sur la pile, saut vers le code de la méthode, retour au code appelant, dépilement des arguments, récupération de la valeur de retour) par une copie du code du corps de la méthode (inline call).
  - si le corps de la méthode est trop gros, le compilateur est censé ne pas faire cette optimisation qui serait contrebalancée par l 'augmentation importante de la taille du code.
  - Mieux vaut ne pas trop se reposer sur le compilateur :
    - utiliser final que lorsque le code n'est pas trop gros ou lorsque l'on veut explicitement éviter toute redéfinition
- méthodes `private` sont implicitement `final` (elles ne peuvent être redéfinies)

Constructeurs et appel de méthodes

```java
public class ClassA {


    public ClassA() {
        foo();
    }

    public void foo() {
        System.out.println("Hello");
    }
}
```

```java
public class ConstructorCallsOverride {

    public static void main(String[] args) {
        new ClassB("world");
    }
}
```

??

```java
public class ClassB extends ClassA {

    protected String s;

    @Override
    public void foo() {
        super.foo();
        System.out.println(s + "(length : " + s.length() + ")");

    }

    public ClassB(String s) {
        this.s = s;
    }
}
```

Constructeurs et appel de méthodes

```java
public class ClassA {


    public ClassA() {
        foo();
    }

    public void foo() {
        System.out.println("Hello");
    }
}
```

```java
public class ConstructorCallsOverride {

    public static void main(String[] args) {
        new ClassB("world");
    }
}
```

```
Hello
Exception in thread "main" java.lang.NullPointerException
        at ClassB.foo(ClassB.java:9)
        at ClassA.<init>(ClassA.java:6)
        at ClassB.<init>(ClassB.java:13)
        at ConstructorCallsOverride.main(ConstructorCallsOverride.java:5)
Java Result: 1
```

```java
public class ClassB extends ClassA {

    protected String s;

    @Override
    public void foo() {
        super.foo();
        System.out.println(s + "(length : " + s.length() + ")");

    }

    public ClassB(String s) {
        this.s = s;
    }
}
```

Constructeurs et appel de méthodes

```
public class ClassA {

    public ClassA() {
        foo();
    }

    public void foo() {
        System.out.println("Hello");
    }
}
```

```
public class ConstructorCallsOverride {

    public static void main(String[] args) {
        new ClassB("world");
    }
}
```

```
Hello
Exception in thread "main" java.lang.NullPointerException
        at ClassB.foo(ClassB.java:9)
        at ClassA.<init>(ClassA.java:6)
        at ClassB.<init>(ClassB.java:13)
        at ConstructorCallsOverride.main(ConstructorCallsOverride.java:5)
Java Result: 1
```

```
public class ClassB extends ClassA {

    protected String s;

    @Override
    public void foo() {
        super.foo();
        System.out.println(s + "(length : " + s.length() + ")");

    }

    public ClassB(String s) {
        this.s = s;
    }
}
```

Appel implicite `super();`

s n'a pas été initialisée (`null`)
→ `NullPointerException`

(1) appel du constructeur de `ClassB`
(2) appel du constructeur de `ClassA`
(3) appel de la méthode `foo`
(4) c'est la méthode `foo` redéfinie dans `ClassB` qui est exécutée

Constructeurs et appel de méthodes

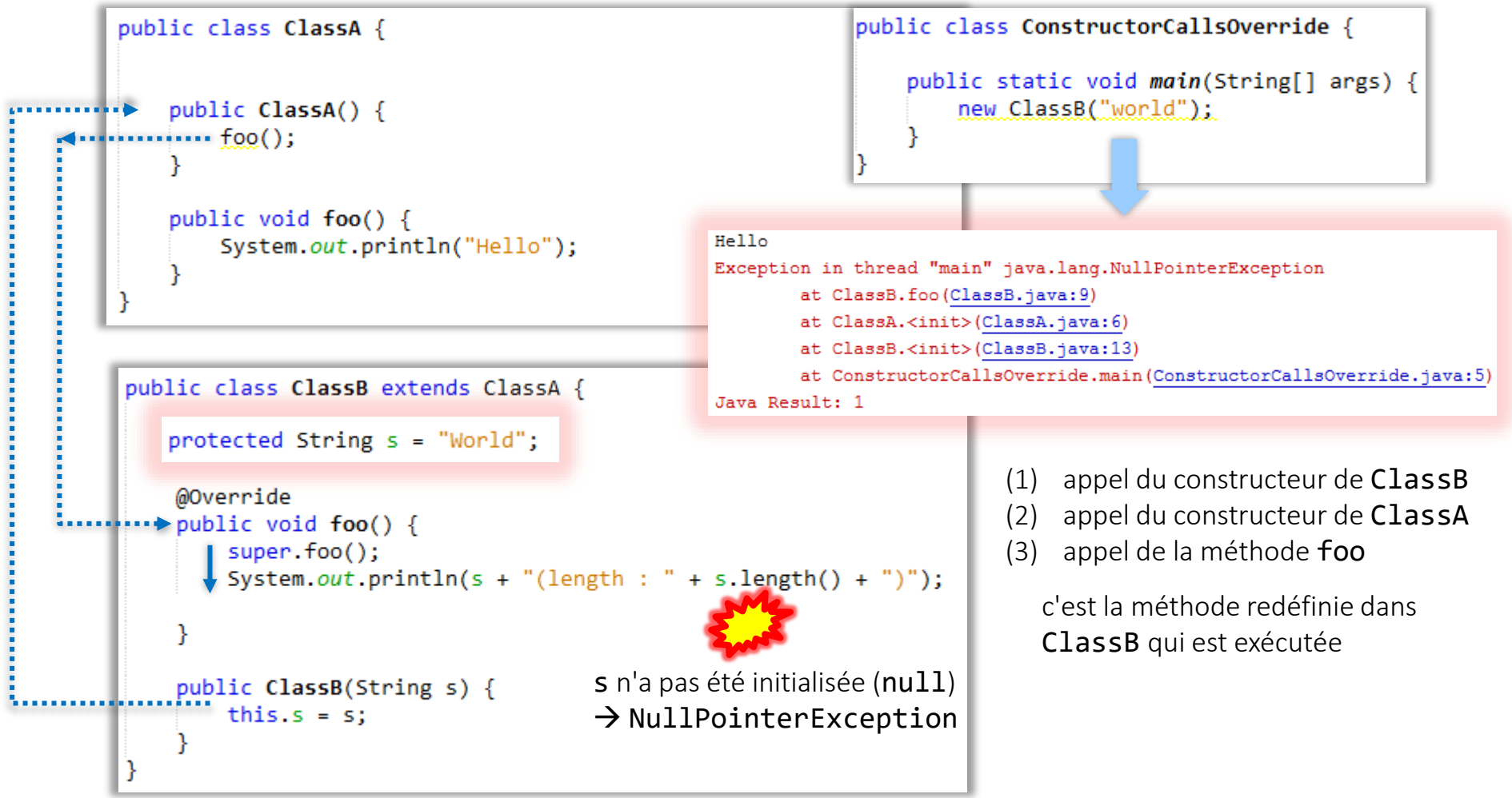
```java
public class ClassA {


    public ClassA() {
        foo();
    }

    public void foo() {
        System.out.println("Hello");
    }
}
```

```java
public class ConstructorCallsOverride {

    public static void main(String[] args) {
        new ClassB("world");
    }
}
```

```
Hello
Exception in thread "main" java.lang.NullPointerException
        at ClassB.foo(ClassB.java:9)
        at ClassA.<init>(ClassA.java:6)
        at ClassB.<init>(ClassB.java:13)
        at ConstructorCallsOverride.main(ConstructorCallsOverride.java:5)
Java Result: 1
```

```java
public class ClassB extends ClassA {

    protected String s = "World";

    @Override
    public void foo() {
        super.foo();
        System.out.println(s + "(length : " + s.length() + ")");

    }

    public ClassB(String s) {
        this.s = s;
    }
}
```

Constructeurs et appel de méthodes



(1) appel du constructeur de `ClassB`
(2) appel du constructeur de `ClassA`
(3) appel de la méthode `foo`

c'est la méthode redéfinie dans `ClassB` qui est exécutée

Constructeurs et appel de méthodes

```java
public class ClassA {


    public ClassA() {
        foo();
    }

    public void foo() {
        System.out.println("Hello");
    }
}
```

```java
public class ConstructorCallsOverride {

    public static void main(String[] args) {
        new ClassB("world");
    }
}
```

```
Hello
Exception in thread "main" java.lang.NullPointerException
        at ClassB.foo(ClassB.java:9)
        at ClassA.<init>(ClassA.java:6)
        at ClassB.<init>(ClassB.java:13)
        at ConstructorCallsOverride.main(ConstructorCallsOverride.java:5)
Java Result: 1
```

```java
public class ClassB extends ClassA {

    protected String s = "World";

    @Override
    public void foo() {
        super.foo();
        System.out.println(s + "(length : " + s.length() + ")");

    }

    public ClassB(String s) {
        this.s = s;
    }
}
```

ne change rien, lorsque **foo** est appelée, **s** n'a pas encore été initialisée

Les variables sont créées à l'allocation (new) et initialisées avec des valeurs nulles.

Les initialiseurs sont exécutés après l'exécution des initialiseurs et du constructeur de la super classe.

Constructeurs et appel de méthodes

Pour que cette instruction soit sans risque il faudrait que la méthode `foo` soit déclarée finale

`final`

```
public class ClassA {


    public ClassA() {
        foo();
    }

    public void foo() {
        System.out.println("Hello");
    }
}
```

```
public class ConstructorCallsOverride {

    public static void main(String[] args) {
        new ClassB("world");
    }
}
```

```
Hello
Exception in thread "main" java.lang.NullPointerException
        at ClassB.foo(ClassB.java:9)
        at ClassA.<init>(ClassA.java:6)
        at ClassB.<init>(ClassB.java:13)
        at ConstructorCallsOverride.main(ConstructorCallsOverride.java:5)
Java Result: 1
```

Mais alors `ClasseB` ne peut pas redéfinir la méthode `foo`

```
public class ClassB extends ClassA {

    protected String s = "World";

    @Override
    public void foo() {
        super.foo();
        System.out.println(s + "(length : " + s.length() + ")");
    }

    public ClassB(String s) {
        this.s = s;
    }
}
```

ne change rien, lorsque `foo` est appelée, `s` n'a pas encore été initialisée

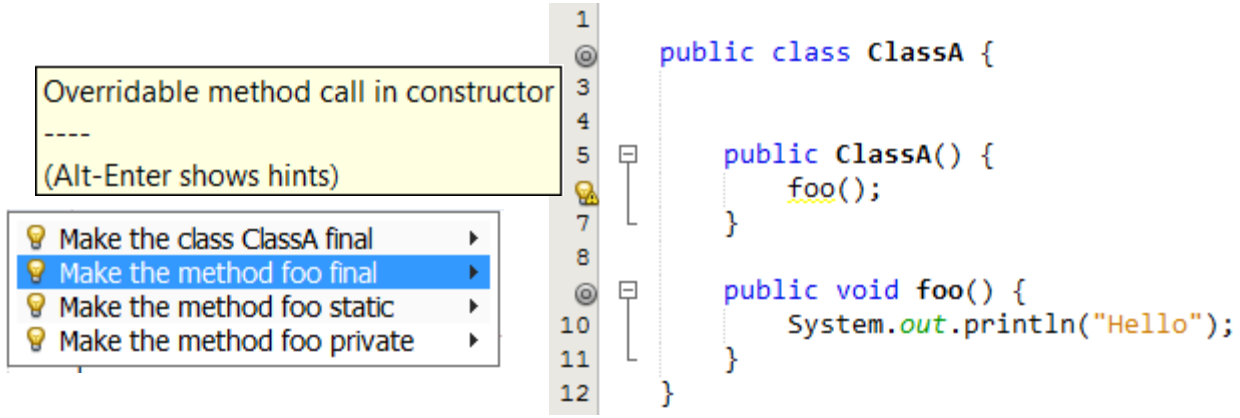
Les variables sont créées à l'allocation (new) et initialisées avec des valeurs nulles.

Les initialiseurs sont exécutés après l'exécution des initialiseurs et du constructeur de la super classe.

"**Constructors must not invoke overridable methods**, directly or indirectly. If you violate this rule, program failure will result. The superclass constructor runs before the subclass constructor, so the overriding method in the subclass will be invoked before the subclass constructor has run. If the overriding method depends on any initialization performed by the subclass constructor, the method will not behave as expected."

Effective Java 2nd Edition, Joshua Bloch, Ed. Addison-Wesley Professional - 2008
Item 17: Design and document for inheritance, or else prohibit it

- en général, une méthode appelée depuis un constructeur doit être déclarée comme finale, sinon une sous classe pourrait la redéfinir avec des résultats indésirables.



Classes finales

```
public final class UneClasse  {

    ...

}
```

- Une classe peut être définie comme finale
  - *interdit tout héritage pour cette classe qui ne pourra être sous-classée*
  - *toutes les méthodes à l'intérieur de la classe seront implicitement finales (elles ne peuvent être redéfinies)*
  - *exemple : la classe* `String` *est finale*

- Attention à l'usage de `final`, prendre garde de ne pas privilégier une supposée efficacité au détriment des éventuelles possibilités de réutiliser la classe par héritage.

Classes scellées (sealed)

- Depuis Java SE 17 possibilité de limiter les possibilités d'héritage sans l'interdire complètement
- classe **scellée** : on indique explicitement quels sont les types de données (classes) autorisés à hériter (dériver) de cette classe

mot clé contextuel introduisant la liste des types dérivés autorisés

liste des sous classes autorisées

mot clé contextuel indiquant que la classe est scellée

```
import java.time.LocalDate;

public sealed class Etudiant permits EtudiantSportif, EtudiantEtranger {
    private String nom;
    private String prenom;
    private LocalDate dateNaissance;

    ...
}
```

Classes scellées (sealed)

- Seule les classes définies dans la liste **`permits`** peuvent être définies

```
import java.time.LocalDate;

public sealed class Etudiant permits EtudiantSportif, EtudiantEtranger {
        ...
}
```

```
import java.time.LocalDate;

public final class EtudiantSportif extends Etudiant {

    String sport;

    ...

}
```

Compile

```
> javac EtudiantSportif.java

>
```

```
import java.time.LocalDate;

public final class EtudiantErasmus extends Etudiant {

    String pays;

    ...

}
```

Erreur de compilation

```
> javac EtudiantErasmus.java
EtudiantErasmus.java:3: error: class is not allowed to extend
sealed class: Etudiant (as it is not listed in its 'permits'
clause)
public final class EtudiantErasmus extends Etudiant {
                   ^
1 error
```

Classes scellées (sealed)

- Lorsque l'on définit une sous classe d'un classe scellée trois alternatives sont proposées
  - *définir une sous classe finale*
  - *définir explicitement une classe ouverte (non scellée)*
  - *définir une classe scellée*
- erreur de compilation si l'on utilise pas une de ces trois alternatives

```
import java.time.LocalDate;

public sealed class Etudiant permits EtudiantSportif, EtudiantEtranger {
        ...
}
```

```
import java.time.LocalDate;

public final class EtudiantSportif extends Etudiant {

    String sport;
    ...
}
```

```
> javac EtudiantSportif.java

>
```

Compile

```
import java.time.LocalDate;

public class EtudiantEtranger extends Etudiant {

    String sport;
    ...
}
```

```
> javac EtudiantEtranger.java
EtudiantEtranger.java:3: error: sealed, non-sealed or
final modifiers expected
public class EtudiantEtranger extends Etudiant {
       ^
1 error
```

Erreur de compilation

Classes scellées (sealed)

- les trois alternatives pour les sous classes d'un classe scellées

```
import java.time.LocalDate;

public sealed class Etudiant permits EtudiantSportif, EtudiantEtranger {
        ...
}
```

```
import java.time.LocalDate;

public final class EtudiantEtranger extends Etudiant {
   ...
}
```

- sous classe finale
  - finaliser la classe interdit de créer une sous classe qui hériterait indirectement de la classe **`Etudiant`**

```
import java.time.LocalDate;

public non-sealed class EtudiantEtranger extends Etudiant {
   ...
}
```

- sous classe ouverte (non scellée)
  - indique au compilateur que toutes les sous classes de la classe **`EtudiantEtranger`** sont autorisées

```
import java.time.LocalDate;

public sealed class EtudiantEtranger extends Etudiant
      permits EtudiantErasmus, EtudiantHorsUE {
   ...
}
```

- sous classe scellée
  - indique au compilateur que toutes les sous classes de la classe **`EtudiantEtranger`** sont autorisées

Classes scellées (sealed)

- Possibilité de scellée d'autres types de données que de simples classes

  - *classes abstraites*

```
public sealed abstract class Forme permits Rectangle, Cercle {
    ...
    public abstract double surface();
    public abstract double perimètre();
}
```

  - *interfaces*

```
public sealed interface Forme permits Rectangle, Cercle {

    double surface();
    double perimètre();
}
```