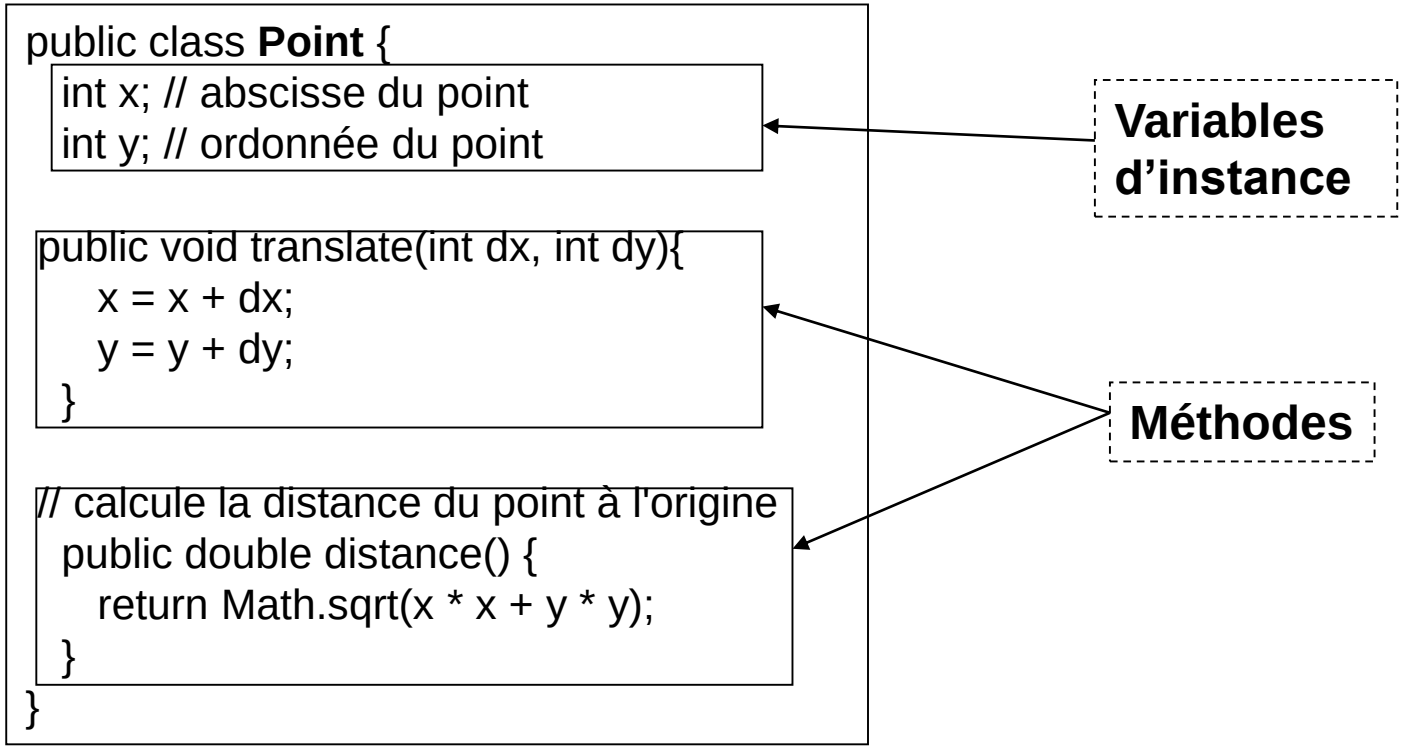
Java
Réutilisation des classes:

- Délégation
- Héritage

Préparé par Larbi Hassouni

Classes : Rappels

- Une classe représente une « famille » d'objets ayant en commun un ensemble de propriétés et comportements.
- Une classe sert à définir les propriétés et comportements des objets d'un type donné.
  - *Elle décrit l 'ensemble des données (attributs,ou **variables**) et des opérations sur données (**méthodes**)*
  - Elle *Sert de « modèle » pour la création d'objets (**instances** de la classe)*

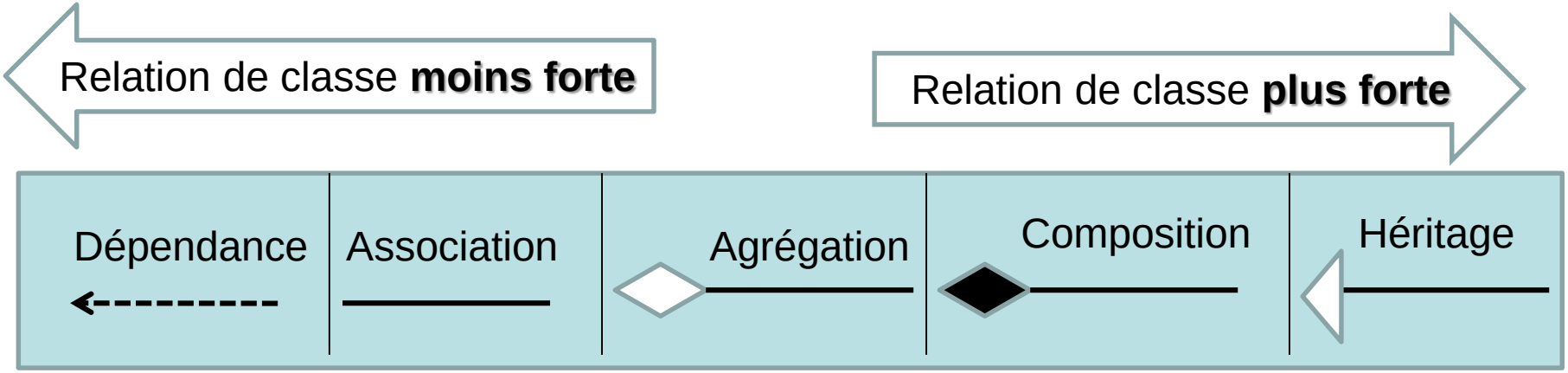


Réutilisation du code : Introduction

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

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

- UML (Unified Modelling Language) définit 5 types des associations possibles entre classes.



Dépendance : Un objet d'une classe **travaille brièvement avec** des objets d'une autre classe

Association : Un objet d'une classe **travaille avec** des objets d'une autre classe pendant une durée prolongée

Agrégation : Une classe **détient et partage une référence** à des objets d'une autre classe

Composition : Une classe **contient** des objets d'une autre classe

Héritage : Une classe **est un** type d'une autre classe

Délégation

Mise en œuvre

Exemple : la classe Cercle

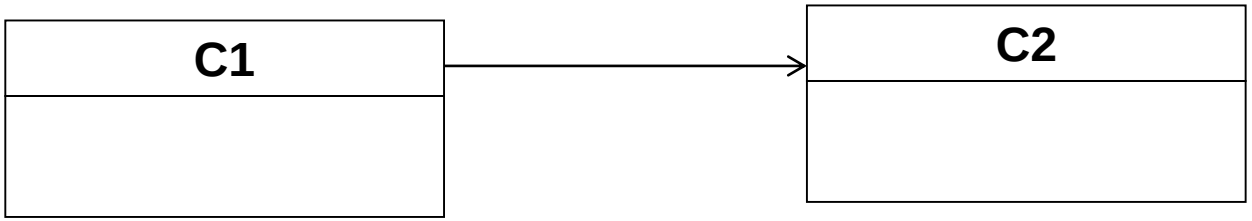
Agrégation / Composition

Délégation : Définition

- L'association, l'agrégation, et la composition sont des fois appelées Délégation

- Un objet o1 instance de la classe C1 utilise les services d'un objet o2 instance de la classe C2 (o1 délègue une partie de son activité à o2)

- La classe C1 utilise les services de la classe C2
  - *C1 est la classe cliente*
  - *C2 est la classe serveuse*

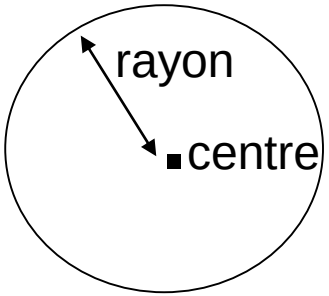


La classe cliente (C1) possède une référence de type de la classe serveuse(C2)

```
public class C1 {

   private C2 champ;
   ...
}
```

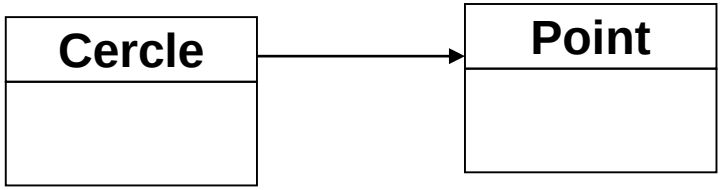
```
public class C2 {

...
}
```

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)** centre

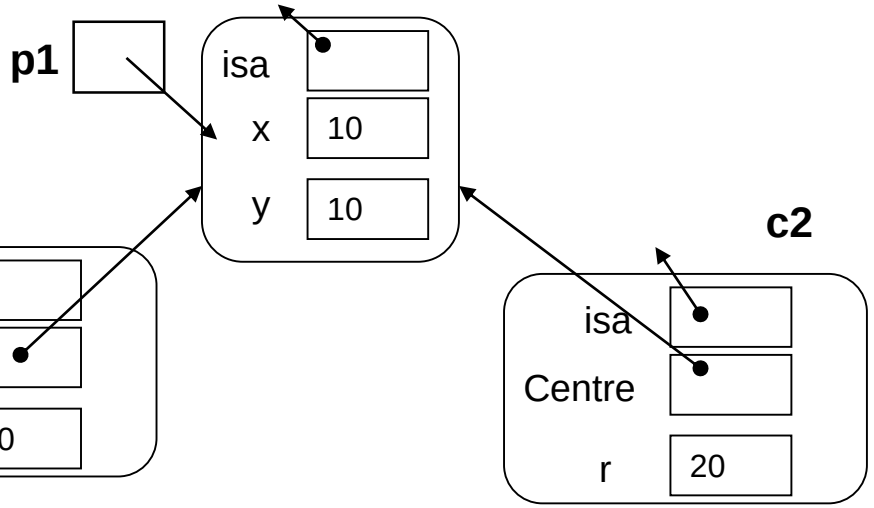
```
public class Cercle {

   // centre du cercle
   private Point centre;

   // rayon du cercle
   private double r;
   …
   public void translation(double dx, double dy) {
      centre.translate(dx,dy);
   }
   …
}
```

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;
   }
   …
}
```

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

- Le point représentant le centre a une existence autonome (cycles de vie indépendants)
- il peut être partagé (à un même moment il peut être liée à plusieurs instances d'autres classes) .



Le point P1 peut être utilisé en dehors du cercle dont il est le centre (Attention aux effets de bord)

```
p1.rotation(90);
c2.translater(10,10);
```

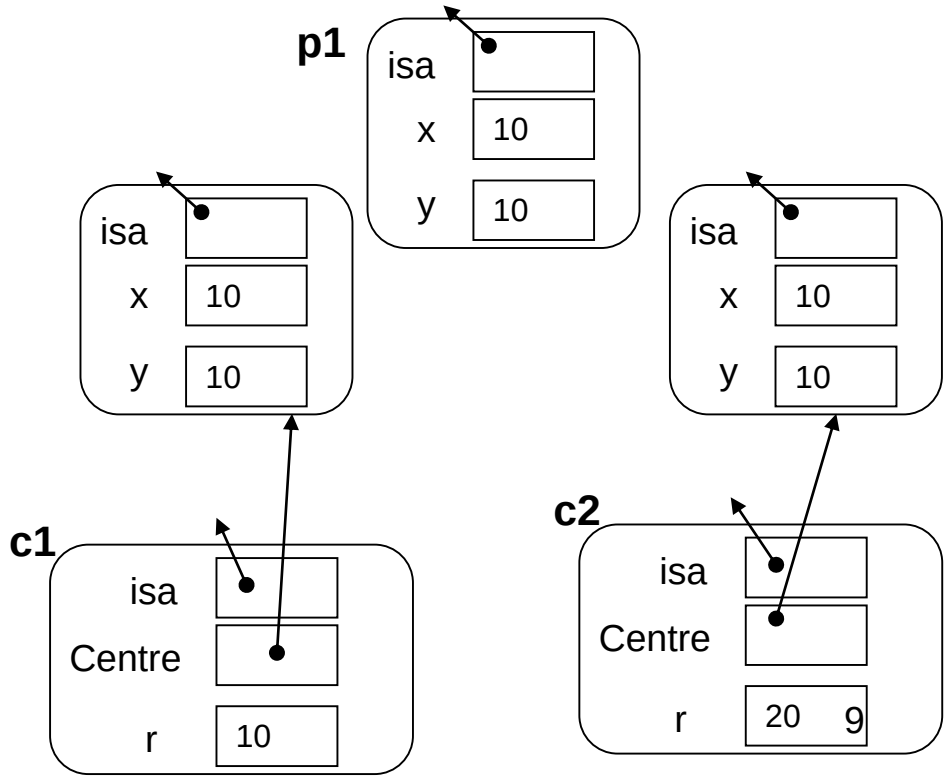
⟹ **Affecte le cercle c1**
Après ces opérations le centre de c1 et c2 devient (0,20)

Délégation : Exemple

```
public class Cercle {
   //centre du cercle
   private Point centre;

   //rayon du cercle
   private double r;

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

- 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(10,10);
Cercle c1 = new Cercle(p1,10)
Cercle c2 = new Cercle(p1,20);
```

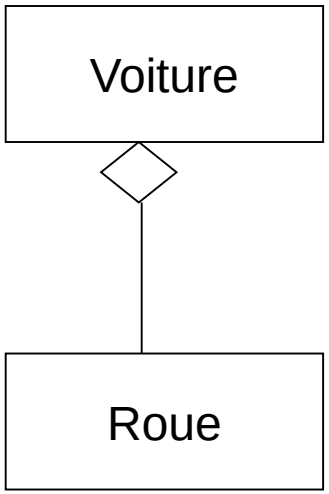


- Les cycles de vies du Point et du Cercle sont liés : si le cercle est détruit (ou copié), le centre l'est aussi.

Agrégation / Composition

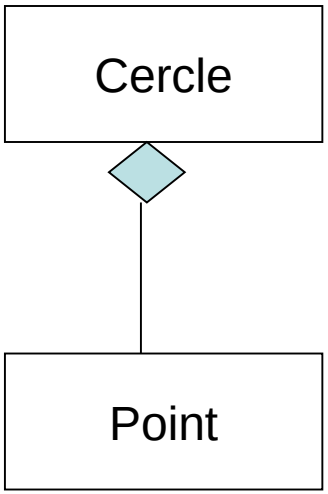
- Les deux exemples précédents traduisent deux nuances (sémantiques) de l'association **a-un** entre la classe Cercle et la classe Point
- UML distingue ces deux sémantiques en définissant deux type de relations :

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

- 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

Réutilisation du code avec modification

- Soit une classe **`A`** dont on a le code compilé
- Une classe **`C`** veut réutiliser la classe **`A`**
- Elle peut créer des instances de **`A`** et leur demander des services
- On dit que la classe **`C`** est une classe cliente de la classe **`A`** : **`Ceci s'appelle la délégation comme on l'a vu précédemment`**
- Souvent, cependant, on souhaite modifier en partie le comportement de **`A`** avant de le réutiliser
- Le comportement de **`A`** convient, sauf pour des détails qu'on aimerait changer
- Ou alors, on aimerait ajouter une nouvelle fonctionnalité à **`A`**

Réutilisation avec modifications du code source

- On peut copier, puis modifier le code source de **`A`** dans des classes **`A1`**, **`A2`**,...

- Problèmes :

  – on n'a pas toujours le code source de **`A`**

  – les améliorations futures du code de **`A`** ne seront pas dans les classes **`A1`**, **`A2`**,…

La solution en POO est l'HERITAGE

définir une nouvelle classe à partir de la classe déjà existante

Réutilisation par l'héritage

- L'héritage existe dans tous les langages objet à classes

- L'héritage permet d'écrire une classe **`B`**
  - qui se comporte dans les grandes lignes comme la classe **`A`**
  - mais avec quelques différences

  sans toucher au code source de **`A`**

- On a seulement besoin du code compilé de **`A`**

- Le code source de **`B`** ne comporte que ce qui a changé par rapport au code de **`A`**
- On peut par exemple
  - ajouter de nouvelles méthodes
  - modifier certaines méthodes

Héritage : classe mère
Exemple 1

```
import java.awt.Graphics;
import java.awt.Color;
public class Rectangle {
  private int x, y; // point en haut à gauche
  private int largeur, hauteur;
  public int  getX(){
     return x;
  }
  public void setX(int x){
     this.x = x;
  }
  // Idem pour getHauteur(), setHauteur(int),
  // getLargeur(), setLargeur(int),
  . . .
  public void dessineToi(Graphics g) {
    g.drawRect(x, y, largeur, hauteur);
  }
}
```

Héritage : classe fille
Exemple 1

```
public class RectangleColore extends Rectangle {
  private Color couleur; // nouvelle variable
  // Constructeurs
  . . .
  // Nouvelles Méthodes
  public getCouleur() {
    return this.couleur;
  }
  public setCouleur(Color c) {
    this.couleur = c;
   }
  // Méthodes modifiées
  public void dessineToi(Graphics g) {
    g.setColor(couleur);
    g.fillRect(getX(), getY(),
               getLargeur(), getHauteur());
  }
}
```

RectangleColore **hérite** de (**étend**) Rectangle. Elle possède les variables et méthodes définies dans la classe Rectangle

Définition d'un nouvel attribut

RectangleColore définit 2 nouvelles méthodes

RectangleColore redéfinit une méthode de la classe mère.

Héritage : Exemple 2

Point.java : Classe mère

```
public class Point {
   protected int x; // abscisse du point
   protected int y; // ordonnée du point

 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);
   }
}
```

PointGraphique.java : Classe fille

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

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

Héritage : Exemple 3

```
// ! c06:Detergent.java
// Syntaxe d'héritage & propriétés.

class Cleanser {
  private String s = new String("Cleanser");
  public void append(String a) { s += a; }
  public void dilute() { append(" dilute()"); }
  public void apply() { append(" apply()"); }
  public void scrub() { append(" scrub()"); }
  public void print() { System.out.println(s); }
  public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    x.print();
  }
}
```

```
public class Detergent extends Cleanser {
  // Change une méthode:
  public void scrub() {
    append(" Detergent.scrub()");
    super.scrub(); // Appel de la version de
la classe de base
  }
  // Ajoute une méthode à l'interface:
  public void foam() { append(" foam()"); }
  // Test de la nouvelle classe:
  public static void main(String[] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    x.print();
    System.out.println("Testing base
class:");
    Cleanser.main(args);
  }
}
```

Terminologie

- La classe **`héritée`** A s'appelle une classe mère, classe parente, classe de base ou super-classe

- La classe **`B`** qui hérite de la classe **`A`** s'appelle une classe fille ou sous-classe

  - ***Point*** *est la* ***classe mère*** *et* ***PointGraphique la classe fille****.*
  - *la classe* ***PointGraphique hérite*** *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**

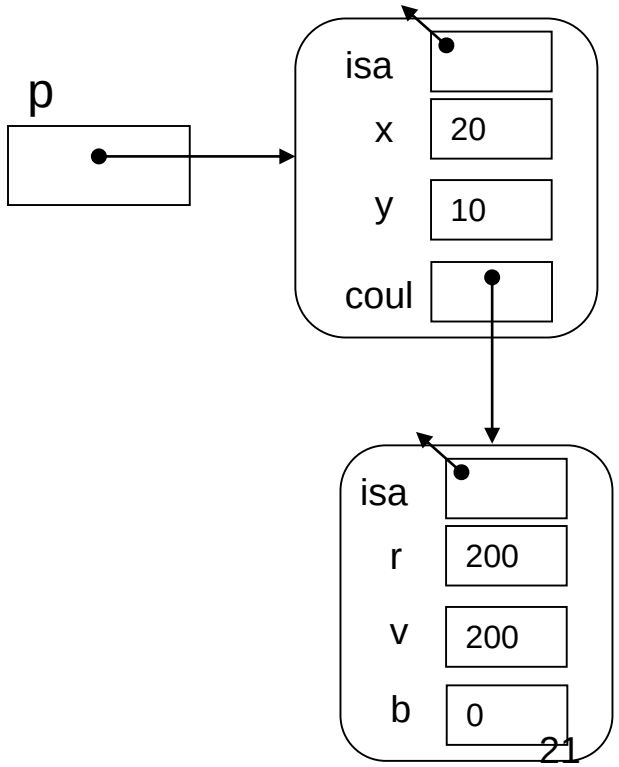
Héritages : Autres exemples

- La classe mère **Vehicule** peut avoir les classes filles **Velo**, **Voiture** et **Camion**
- La classe **Avion** peut avoir les classes mères **ObjetVolant** et **ObjetMotorise**
- La classe **Polygone** peut hériter de la classe **FigureGeometrique**
- La classe **Image** peut avoir comme classe fille **ImageGIF** et **ImageJpeg**
- La classe **Animal** peut avoir comme classe fille **Singe** et **Cheval.**

Utilistaions 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 = 20;
p.y = 10;
// utilisation d'une variable d'instance spécifique
p.coul = new Color(200,200,0);
// utilisation d'une méthode héritée
double dist = p.distance();
// utilisation d'une méthode spécifique
p.dessine(graphicContext);
```



Héritage : Généralisation / Spécialisation

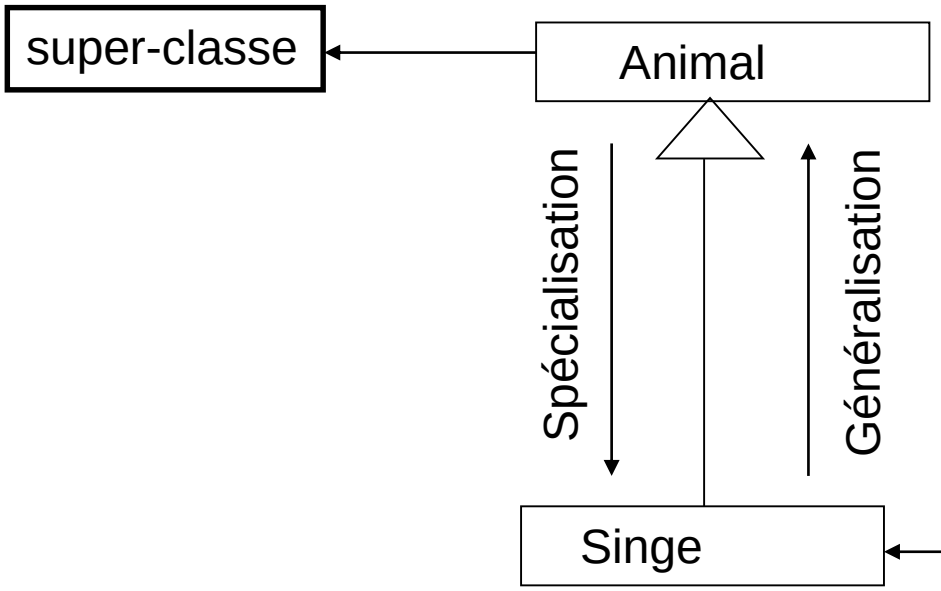
- la relation d'héritage peut être vue comme une relation de "**généralisation/spécialisation**" entre une classe (la *super-classe*) et plusieurs classes plus spécialisées (ses *sous-classes*).

- **Particularisation ou Spécialisation**:
  - un rectangle coloré *est un* rectangle mais un rectangle particulier
  - Un PointGraphique est un point mais un point particulier
- **Généralisation:**
  - la notion de figure géométrique est une généralisation de la notion de polygone
  - Un animal est une généralisation du cheval et du singe

- Une classe fille offre de nouveaux services ou enrichit les services rendus par une classe : la classe **RectangleColore** permet de dessiner avec des couleurs et pas seulement en « noir et blanc »

Héritage : Généralisation / Spécialisation

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



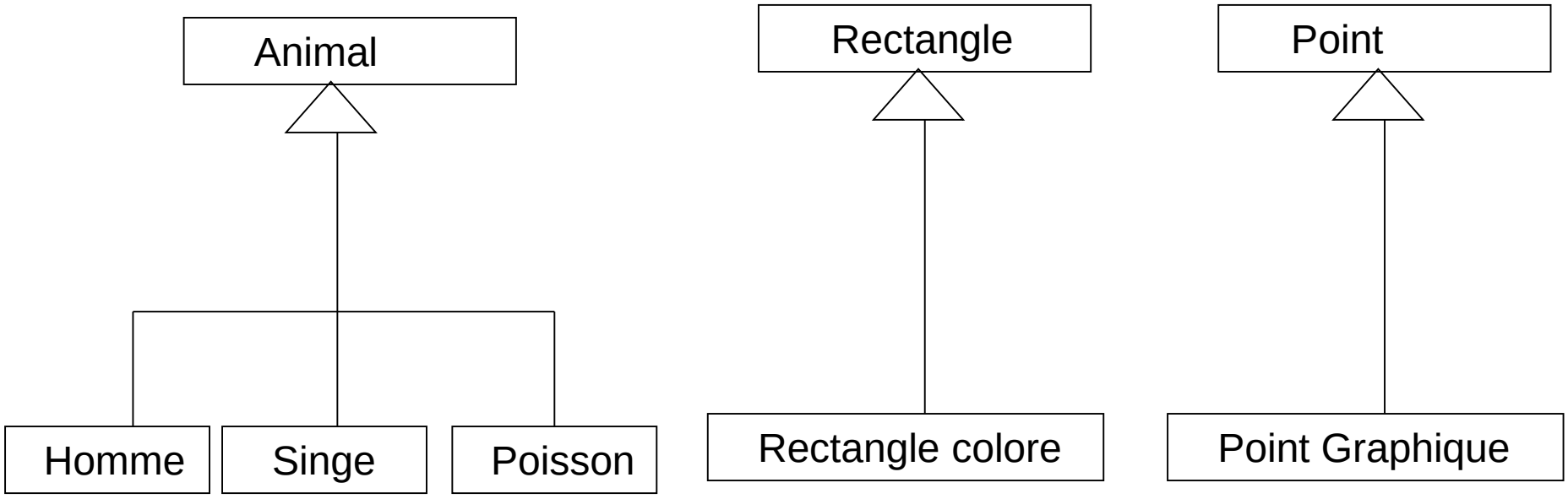
Singe est une **spécialisation** d’**Animal**.

Un singe ***est un cas particulier*** d’animal

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)

Héritage : Généralisation / Spécialisation

- 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,*

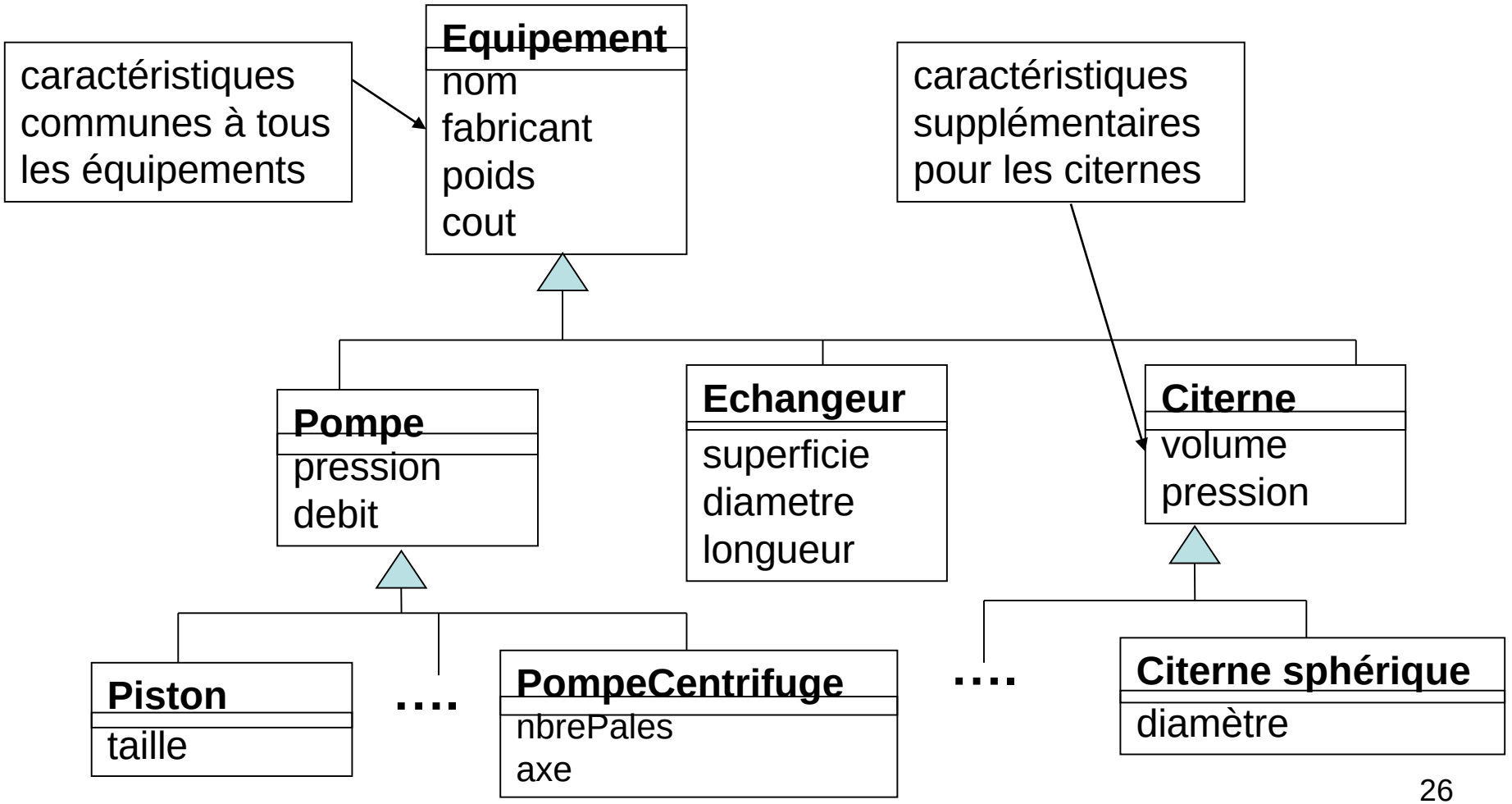


Héritage : Différences entre les langages

- Chaque langage objet a ses particularités :
- Par exemple, C++ permet l'héritage multiple ; par contre Java et C# ne le permettent pas
- Dans la suite du cours nous nous intéresserons à l'héritage dans le langage Java

Héritage à plusieurs niveaux

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



Héritage à plusieurs niveaux : Exemple

```
public class A {
   public void hello() {
      System.out.println(«Vous etes en classe A »);
   }
}

 public class B extends A {
    public void bye() {
       System.out.println(«Vous etes en classe B»);
 }
 }

   public class C extends B {
      public void bof() {
         System.out.println(«Vous etes en classe C»);
      }
   }
```

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

```
C unC = new C();
unC.hello();
unC.bye();
unC.bof();
```

Que peut on mettre dans une classe fille?

- Dans la classe qui hérite, on peut
  - ajouter des variables, des méthodes et des constructeurs
  - redéfinir (override) des méthodes, et fournir ainsi des implémentations spécialisées pour celles-ci.

    (Lorsqu'on redéfinit une méthode dans une classe fille, on doit donner le même nom, , même type de retour, et mêmes types des paramètres).
  - surcharger des méthodes (même nom mais pas même signature)

    (la surcharge est aussi possible à l'intérieur d'une classe)

- Mais
  - elle ne peut retirer aucune variable ou méthode

Lorsqu'une méthode redéfinie par une classe est invoquée pour un objet de cette classe, c'est la nouvelle définition et non pas celle de la super-classe qui est invoquée.

Héritage : redéfinition des méthodes

```
public class A {
   public void hello() {
      System.out.println(«Hello»);
   }
   public void affiche() {
      System.out.println(«vous etes en A»);
   }
}
```

```
public class B extends A {
   public void affiche() {
      System.out.println («vous etes en B»);
   }
}
```

A unA = new A();
B unB1 = new B();

unA.hello(); ➔ Hello
unA.affiche(); ➔vous etes en A

unB1.hello(); ➔ Hello
unB1.affiche(); ➔vous etes en B

A unB2 = new B(); ?
unB2.hello(); ➔ Hello ?
unB2.affiche(); ➔ vous etes en B ?

Voir polymorphisme

Surcharge ou redéfinition de méthodes

- Attention de 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) {
      ...
   }
}
```

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

Redéfinition

```
public class C extends A {
   public void methodX(int i) {
      ...
   }
}
```

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

Redéfinition avec réutilisation : mot super

- **Redéfinition** des méthodes (method **overriding**) :
  - *Il est possible de réutiliser le code de la méthode héritée (en utilisant le mot* **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);
      ...
   }
}
```

Limite pour désigner une méthode redéfinie

- On ne peut remonter plus haut que la classe mère pour récupérer une méthode redéfinie :
  - pas de *cast* « **(ClasseAncetre)affiche()** »
  - pas de « **super.super.affiche()** »

Remarque : Depuis le JDK 5 on peut accoler une annotation à une méthode qui redéfinit une méthode d'une classe ancêtre : **@Override**

C'est très utile pour repérer des fautes de frappe dans le nom de la méthode : le compilateur envoie un message d'erreur si la méthode ne redéfinit aucune méthode d'une classe ancêtre. Il est donc fortement conseillé de le faire

Racine de la hiérarchie d'héritage : Classe Object

❑**La hiérarchie d'héritage est un arbre dont la racine est la classe Object (package 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 {
    int x; // abscisse du point
    int y; // ordonnée du point
    ...
}
```

≡

```
public class Point extends Object {
    int x; // abscisse du point
    int y; // ordonnée du point
    ...
}
```

Classe **Object**

La classe **Object** n'a pas de variables d'instance ni de variables de classe.

- La classe **Object** fournit plusieurs méthodes qui sont héritées par toutes les classes sans exception
- Les plus couramment utilisées sont les méthodes **toString()** et **equals().**
- **public String toString()**

  Renvoie une chaîne représentant la valeur de l'objet

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

- **public boolean equals(Object obj)**

  Teste l'égalité de l'objet avec l'objet passé en paramètre. (voir polymorphisme)

  return (this == obj);

Autres méthodes:

- **public int hashCode()**

  Renvoie une clé de hashcode pour adressage dispersé (Voir collections)
- **protected Object clone()**

  Crée un copie de l'objet
- **public final Class getClass()**

  Renvoie la référence de l'objet Java représentant la classe de l'objet

Classe **Object : méthode toString**

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

≡

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

Opérateur de concaténation

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

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

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

méthode toString :Utilisation

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

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

Cette classe Point ne redéfinit pas toString

```
public class Point {
   private double x,
   private double y;
   public String toString(){
      return ("Point:[" + x + "," + y + "]");
   }
}
```

Cette classe Point redéfinit toString

```
Point p = new Point(20,30);
System.out.println(p);
```

Point@2b580c

Point:[20,30]

Exemple de **toString, equals et hashCode**

```
public class Fraction {
  private int num, den;
  . . .
  @Override
public String toString() {
    return num + "/" + den;
  }
 @Override
  public boolean equals(Object o) {
    if (! (o instanceof Fraction))
      return false;
    return num * ((Fraction)o).den
        == den * ((Fraction)o).num;
  }
private static int pgcd(int i1, int i2) {
    if(i2 == 0) return i1;
    else return pgcd(i2, i1 % i2);
  }
```

a/b = c/d

si et seulement si

a.d = b.c

Exemple de **toString, equals et hashCode**

```
public Fraction reduire() {
    int d = pgcd(num, den);
    return new Fraction(num/d, den/d);
  }

// Calcul du hashcode : Réduit la fraction,  puis calcule un int
  public int hashCode() {
    Fraction f = reduire();
    return (17 + f.num * 37) * 37 + f.den;
  }
}
```

classe **Class**

- Class est une classe particulière (voir cours sur la reflection)

- Les instances de la classe **java.lang.Class** représentent les classes utilisées par Java (et aussi les types primitifs)
- **public Class getClass()**
  renvoie la classe de l'objet
- La méthode **getName()** de la classe **Class** renvoie le nom complet de la classe (avec le nom du paquetage)

- **Instanceof**
  - ✓ Si **x** est une instance d'une sous-classe **B** de **A**,
    « **x instanceof A** » renvoie **true**

✓ Pour tester si un objet **o** est de la même classe que l'objet courant, il ne faut donc pas utiliser **instanceof** mais le code suivant :
if (o != null && o.getClass() == this.getClass())

Constructeurs
Réutilisation des constructeurs

- Nous avons vu que lors de la redéfinition d'une méthode, il est *possible de réutiliser le code de la méthode héritée à l'aide du mot* **super.**

- De la même manière il est important de pouvoir réutiliser le code des constructeurs de la classe mère dans la définition des constructeurs d'une classe fille.

- L'i*nvocation d'un constructeur de la super classe se fait par l'instruction :*

  super(*paramètres du constructeur*)

- **Attention:** cette instruction doit être la première dans le constructeur d'une classe fille.

- L'utilisation de super(…) est analogue à celle de this.

Réutilisation des constructeurs Exemple 1

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

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

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

Réutilisation des constructeurs
Exemple 2

```
public class Rectangle {
  private int x, y, largeur, hauteur;

  public Rectangle(int x, int y,
                   int largeur, int hauteur) {
    this.x = x;
    this.y = y;
    this.largeur = largeur;
    this.longueur = longueur;
  }
  . . .
}
```

```
public class RectangleColore extends
Rectangle {
  private Color couleur;
  public RectangleColore(int x, int y,
                         int largeur, int hauteur
                         Color couleur) {
    super(x, y, largeur, hauteur);
    this.couleur = couleur;
  }
  public RectangleColore(int x, int y,
                         int largeur, int hauteur) {
    this(x, y, largeur, hauteur, Color.black);
  }
  . . .
}
```

Appel du constructeur de la super classe : doit être la 1ere instruction

Appel d'un autre constructeur de la même classe : doit être la 1ere instruction

Appel implicite du constructeur de la classe mère

- L'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 super classe, 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.*

Appel implicite du constructeur de la classe mère

Il est garanti 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 d'une classe C sous classe de B elle même sous classe de A est un objet de classe C mais est aussi un objet de classe B et de classe A.

Lorsqu'il est créé c doit l'être avec les caractéristiques d 'un objet de A, de B et de C.

Remarque : C'est le constructeur de la classe Object qui crée l'objet en mémoire

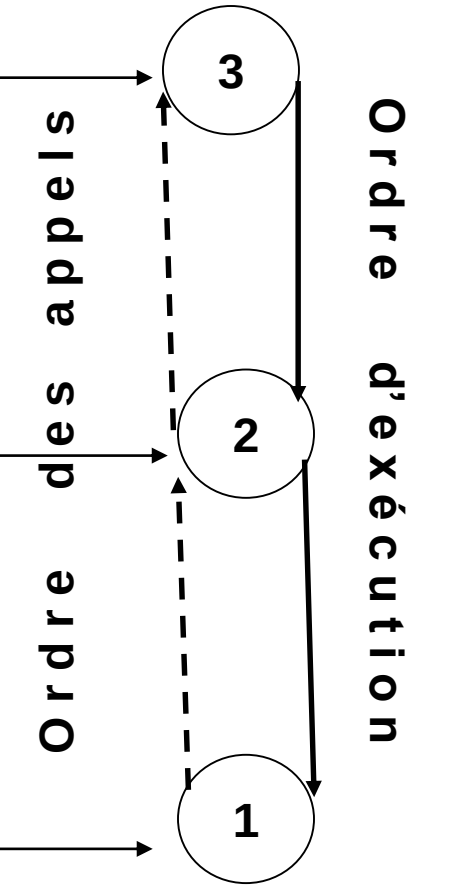
(il est le seul à savoir le faire).

Chaînage des constructeurs

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

```
public class Point  {
   double x,y;
   public Point(double x, double y){
      // Appel implicite du super()
      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;
   }
   ...
}
```



PointCouleur unPC =

new PointCouleur(…..)

Constructeur par défaut

❑ Lorsqu'une classe ne définit pas explicitement de constructeur, elle possède un constructeur par défaut. Ce constructeur a les caractéristiques suivantes:

- Il n'a pas de *paramètres*
- *Son corps est vide**, il contient par défaut l'appel au constructeur super();***
- *Il est masqué dans le cas où un autre constructeur est défini.*

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

```
public class A {
   // attributs
   String nom;

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

Constructeur par défaut implicite

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

Il 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

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

   // pas de constructeur
}
```

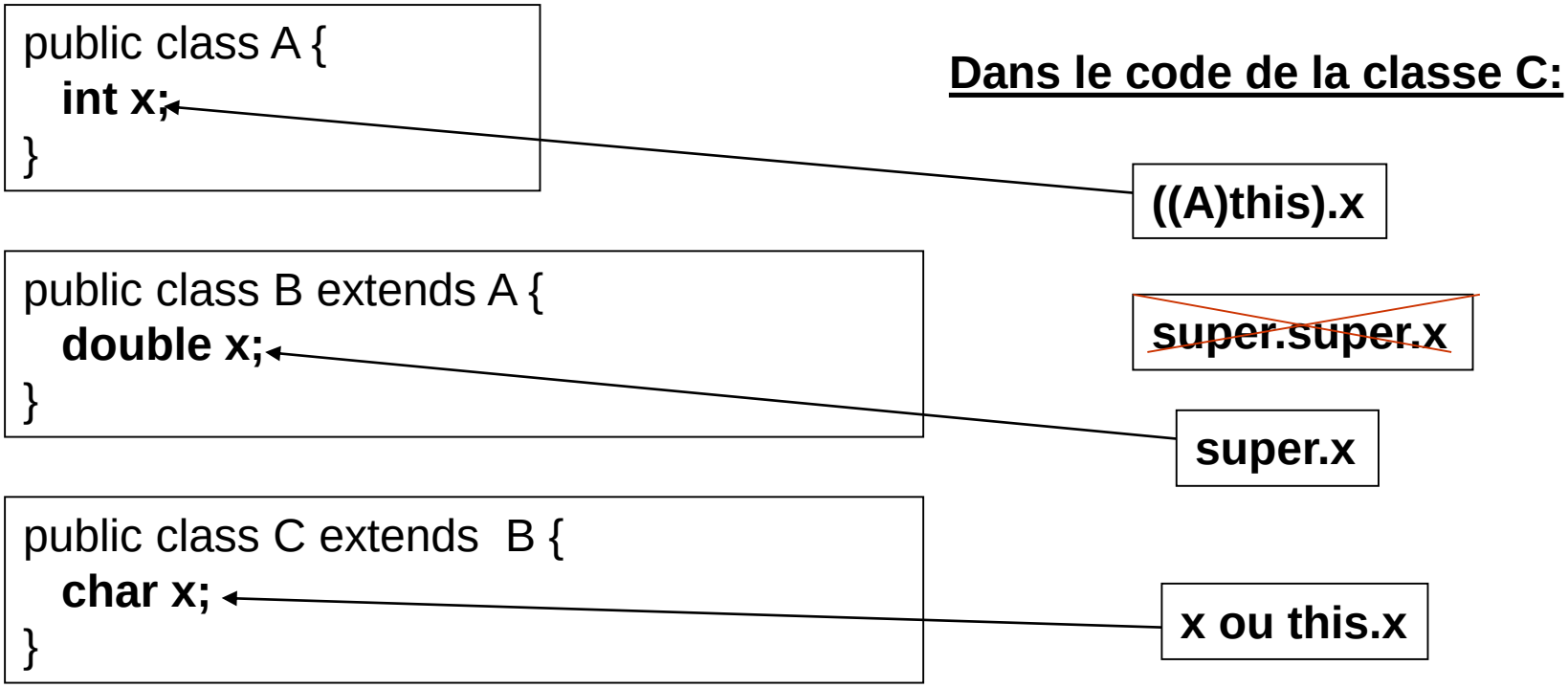
```
public ClasseB(){
   super();
}
```

Constructeur
par défaut implicite

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

Redéfinition des attributs

- 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 en utilisant* **super**. Mais il n'est pas recommandé de masquer les variables.



Redéfinition des attributs: Exemple

```
class A{
  protected int x =5;
  public void affiche(){
    System.out.println("x de A = " +x);
  }
}

class B extends A{
  protected double x = 5.5;
    public void affiche(){
      super.affiche();
      System.out.println("x de B = " +x);

    }
}
```

```
public class C extends B{
  char x = 'c';
  public void affiche(){
    //((A)this).affiche(); Attention ne marche pas
    super.affiche();
    System.out.println("x de C = " +x);
  }
  public void affiche2(){
    System.out.println("x de A " + ((A)this).x);
    System.out.println("x de B " + super.x);
    System.out.println("x de C " + x);
  }
  public static void main(String[] args){
    C unC = new C();
    unC.affiche();

System.out.println("======================");
    unC.affiche2();
  }
}
```

```
x de A = 5
x de B = 5.5
x de C = c
===============
x de A = 5
x de B = 5.5
x de C = c
```

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

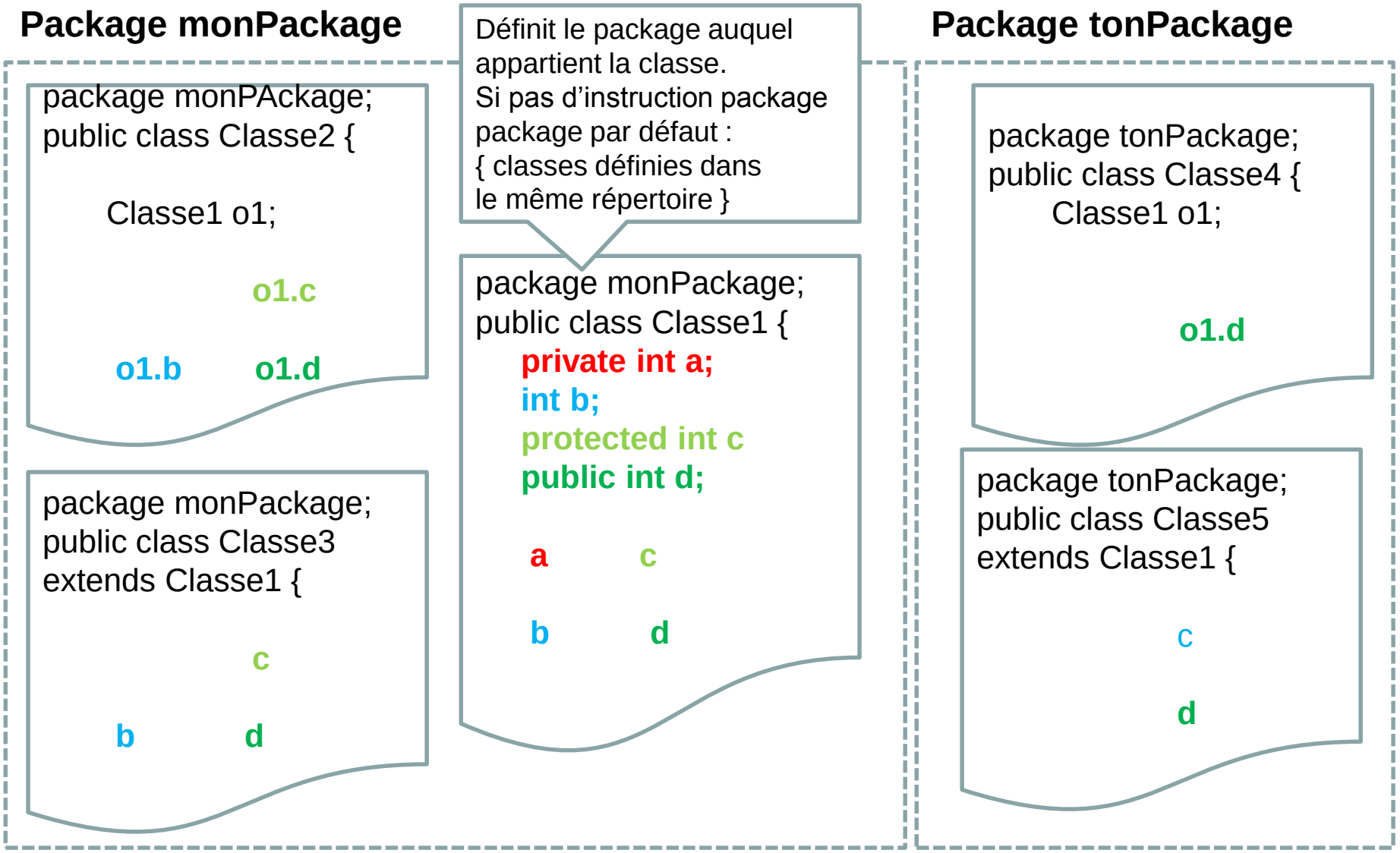
en JAVA possibilité de contrôler l'accessibilité (visibilité) des membres (variables et méthodes) d'une classe

public : *accessible à toute autre classe*
private : *n'est accessible qu'à l'intérieur de la classe où il est défini*
protected: *est accessible dans la classe où il est défini, dans toutes ses sous-classes et dans toutes les classes du même package*
package : *(visibilité par défaut) 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

Visibilité des variables et méthodes



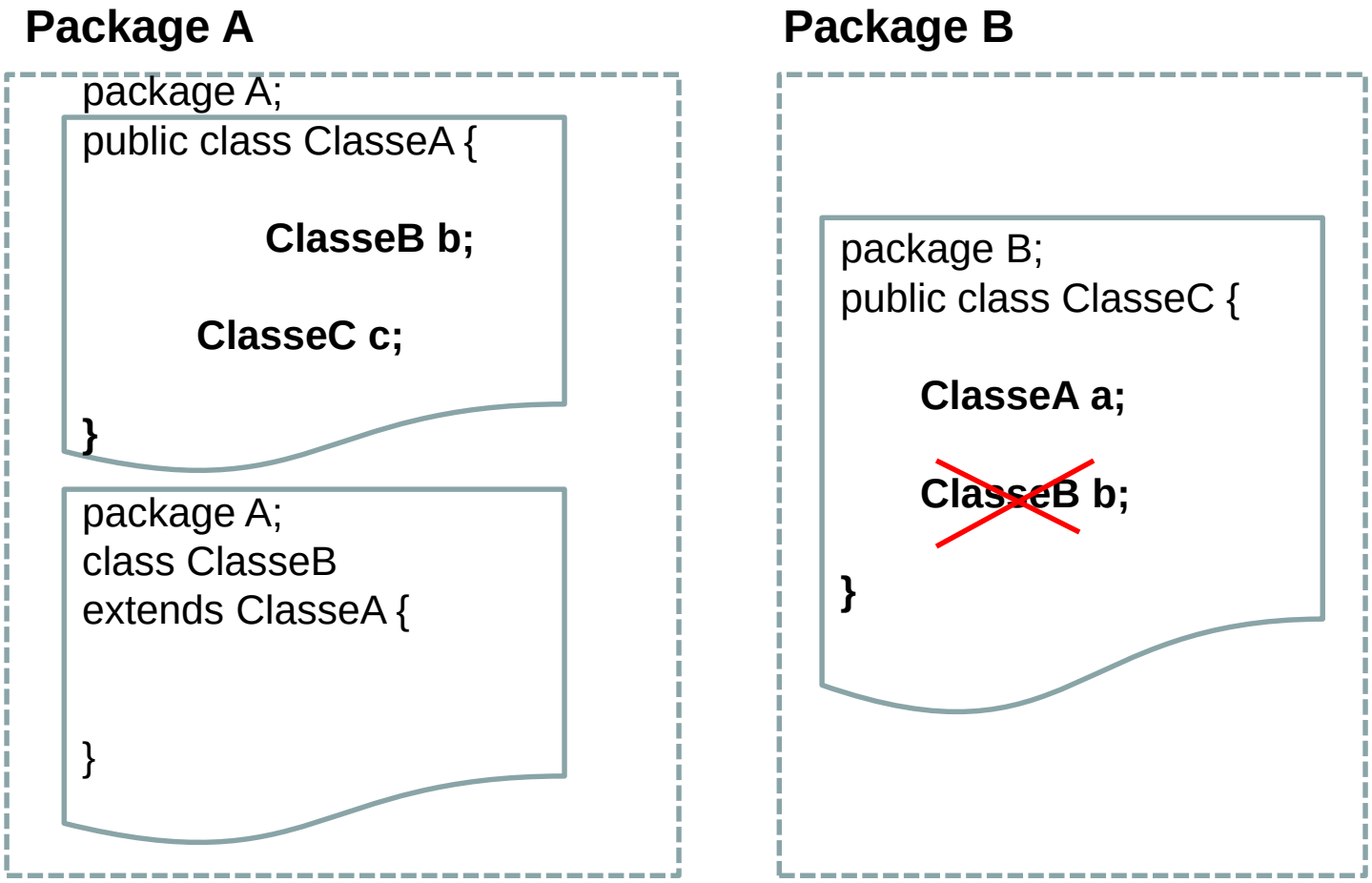
Les mêmes règles de visibilité s'appliquent aux méthodes

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



Méthodes finales

- Méthodes finales
  - ♦ public **final void méthodeX(….) {**
    …. }
  - ♦ *«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)***

Classes finales

- Une classe peut être définie comme finale

```
public final class UneClasse {
        …
}
```

  - ♦ *Interdit tout héritage pour cette classe qui ne pourra être sous-classée*
  - ♦ *toutes les méthodes à l'intérieur de la classe seront implicitiment 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.

Exercice : Implémenter le diagramme des classes ci-dessous: