

Réutilisation des classes

Délégation

Héritage

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 {
```

```
    private 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);  
    }
```

```
}
```

**Variable
de classe**

**Variables
d'instance**

Constructeur

Méthodes

Membres

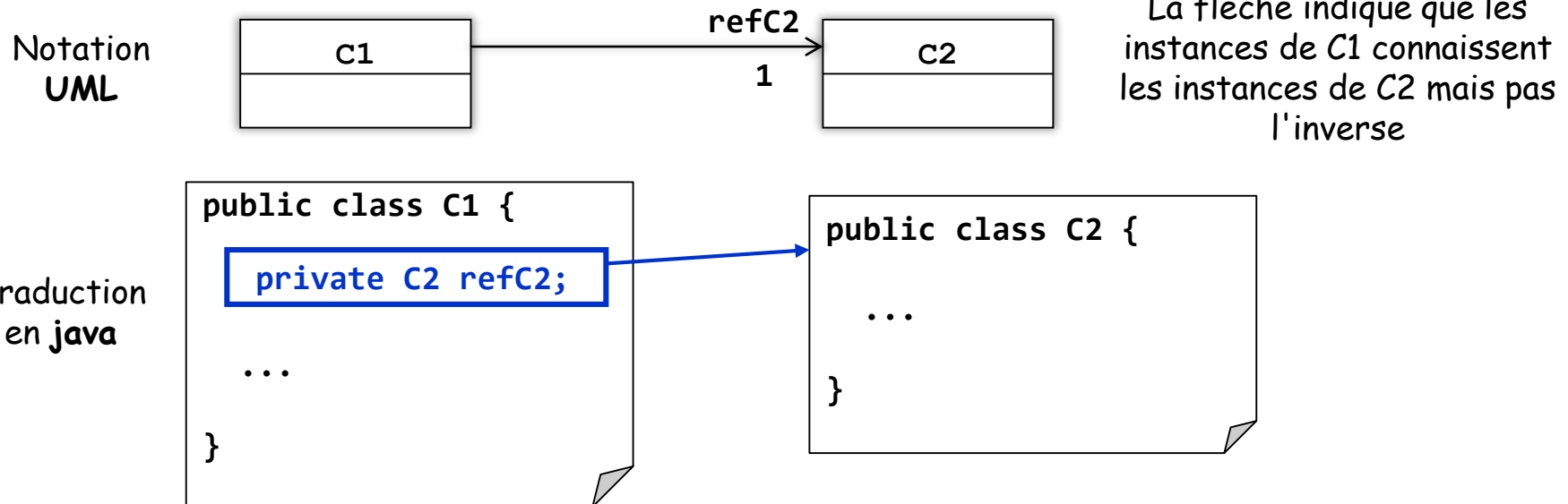
Réutilisation : introduction

- 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>) définit 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

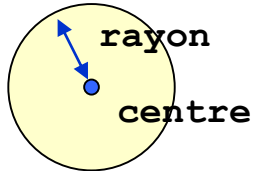
- Un objet **o1** instance de la classe **C1** utilise les services d'un objet **o2** instance de la classe **C2** (**o1** délègue une partie de son activité à **o2**)
- La classe **C1** utilise les services de la classe **C2**
 - *C1 est la classe cliente*
 - *C2 est la classe serveuse*



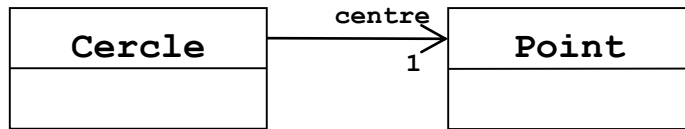
- La classe cliente (**C1**) possède une référence de type de la classe serveuse (**C2**)

Délégation : exemple

- Exemple la classe **Cercle**



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



- L'association entre les classes Cercle et PointCartesien 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;

    ...

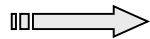
    public void translater(double dx, double dy) {
        centre.translater(dx,dy);
    }    le cercle délègue à son centre l'opération de translation

    ...
}
```

Délégation : exemple

```
public class Cercle {  
  
    /** centre du cercle */  
    private Point centre;  
  
    /** rayon du cercle*/  
    private double r;  
  
    public Cercle( Point centre, double r) {  
        this.centre = centre;  
        this.r = r;  
    }  
    ...  
  
    public void translater(double dx, double dy) {  
        centre.translater(dx,dy);  
    }  
}
```

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



Affecte aussi cercle c1

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

```
p1.rotation(90);
```

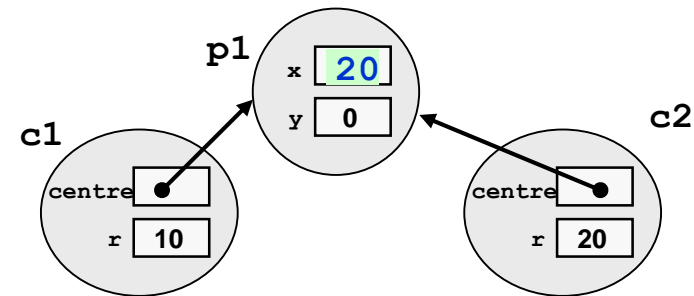


Affecte les deux cercles c1 et c2

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

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

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



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

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;  
    }  
    ...  
}
```

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

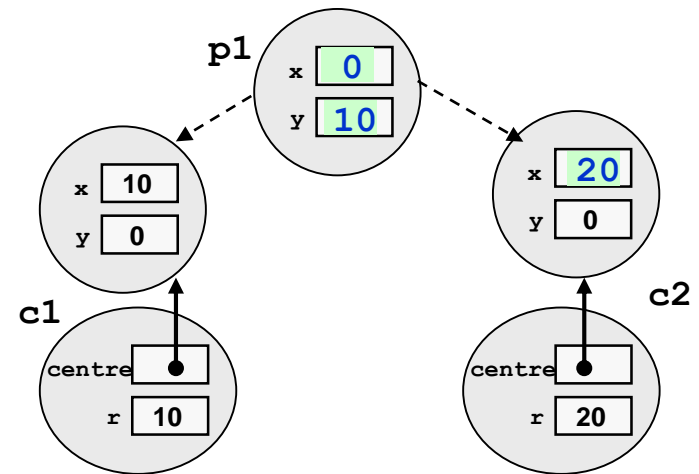
⇒ N'affecte que le cercle c2

```
p1.rotation(90);
```

⇒ N'affecte pas les cercles c1 et c2

- 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);  
...
```

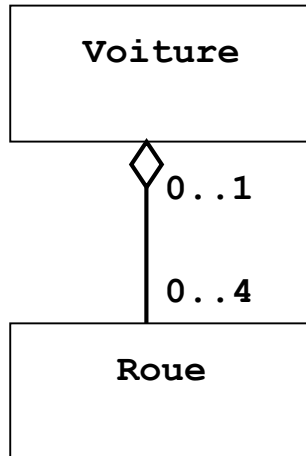


- les cycles de vies du **Point** et du **Cercle** sont liés : si le cercle est détruit (ou copié), le centre l'est aussi.

Agrégation / Composition

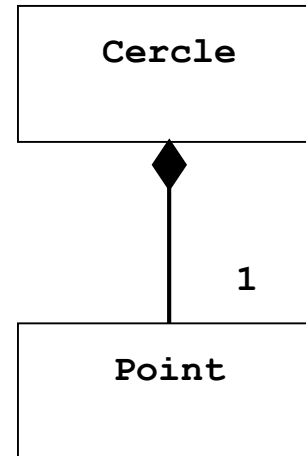
- Les deux exemples précédents traduisent deux nuances (sémantiques) de l'association **a-un** entre la classe **Cercle** et la classe **Point**
- UML distingue ces deux sémantiques en définissant deux type de relations :

Agrégation



L'élément agrégé (**Roue**) a une existence autonome en dehors de l'agréga (**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'agréga.

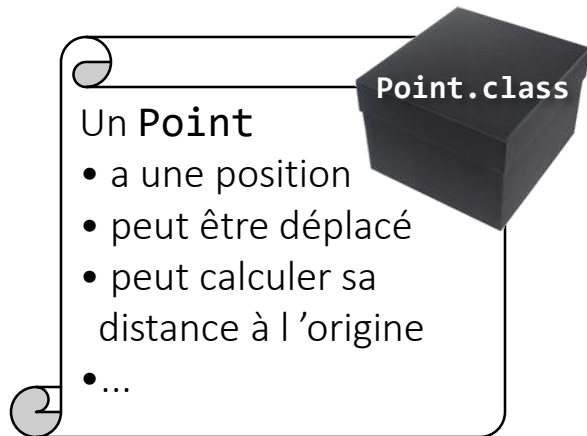
Héritage

- 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

Exemple introductif

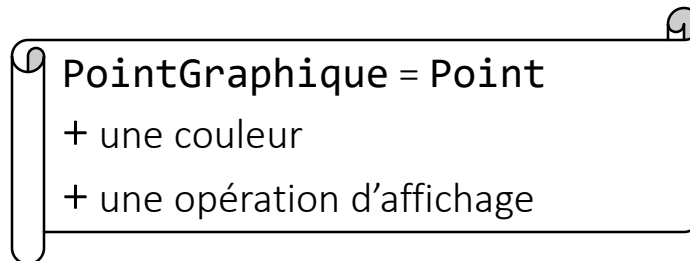
- Le problème

- 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.



- Solution en POO : l'héritage (inheritance)

- définir une nouvelle classe à partir de la classe déjà existante

- La classe `PointGraphique` hérite de la classe `Point`

`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**

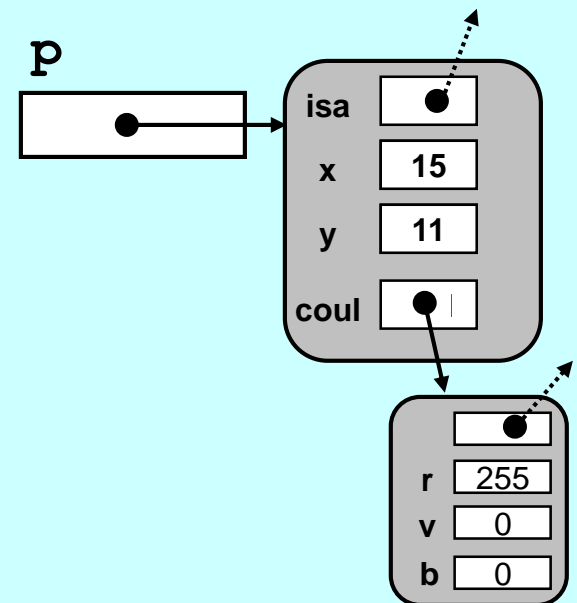
PointGraphique
définit une nouvelle
méthode

Utilisation des instances d'une classe héritée

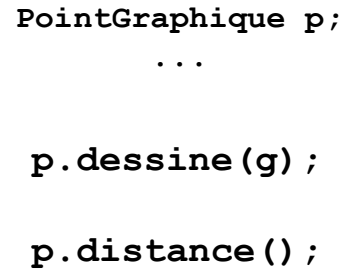
- Un objet instance de *PointGraphique* possède les attributs définis dans *PointGraphique* ainsi que les attributs définis dans *Point* (un *PointGraphique* est aussi un *Point*)
- Un objet instance de *PointGraphique* répond aux messages définis par les méthodes décrites dans la classe *PointGraphique* 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);
```

mémoire



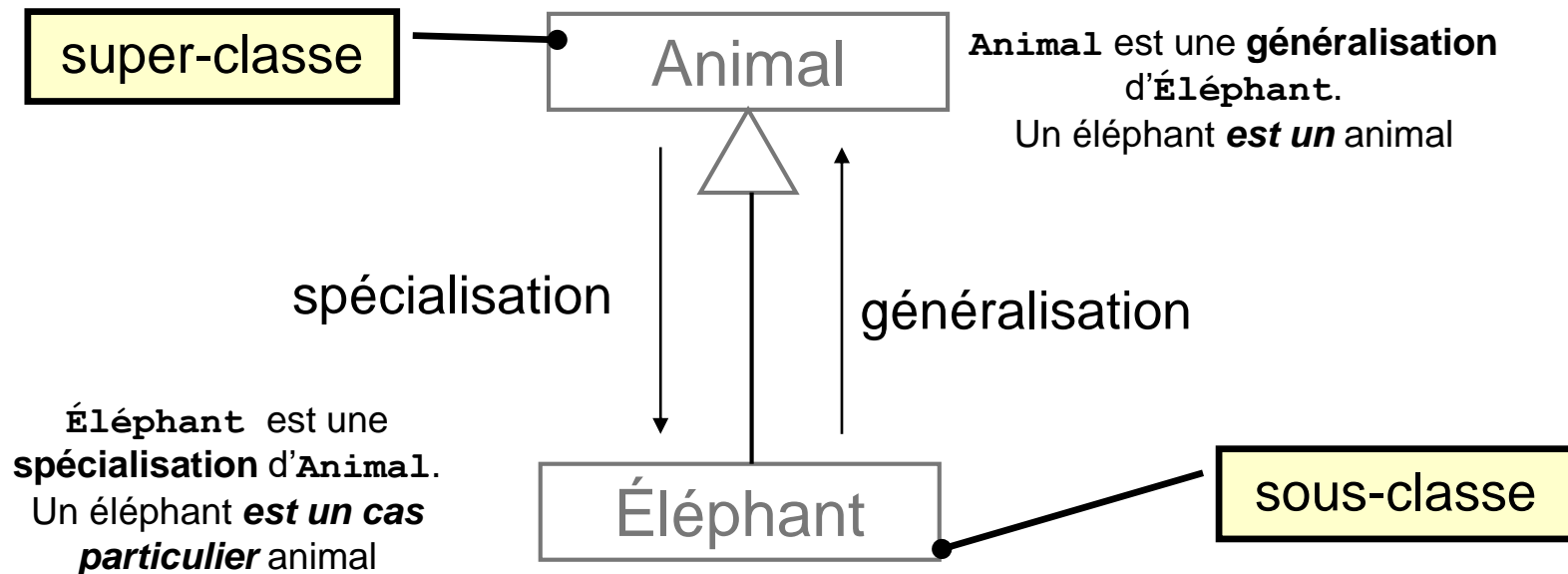
résolution des messages



- **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 de **F** possèdent toutes les caractéristiques de **M** avec 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 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*).

Généralisation/Spécialisation

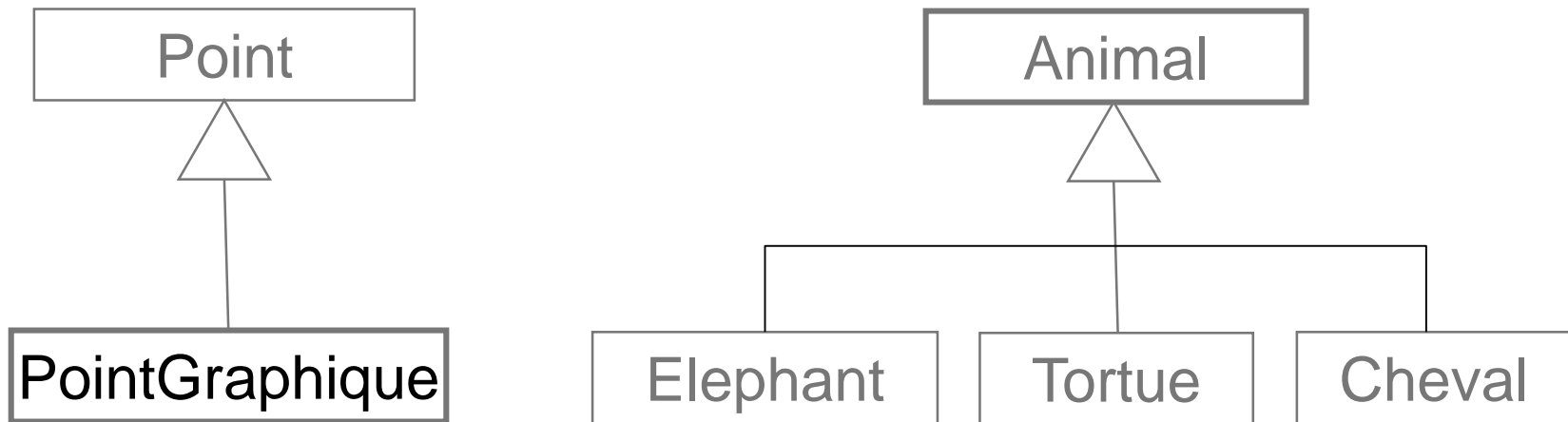
- La généralisation exprime une relation “**est-un**” entre une classe et sa super-classe (chaque instance de la classe est aussi décrite de façon plus générale par la super-classe).



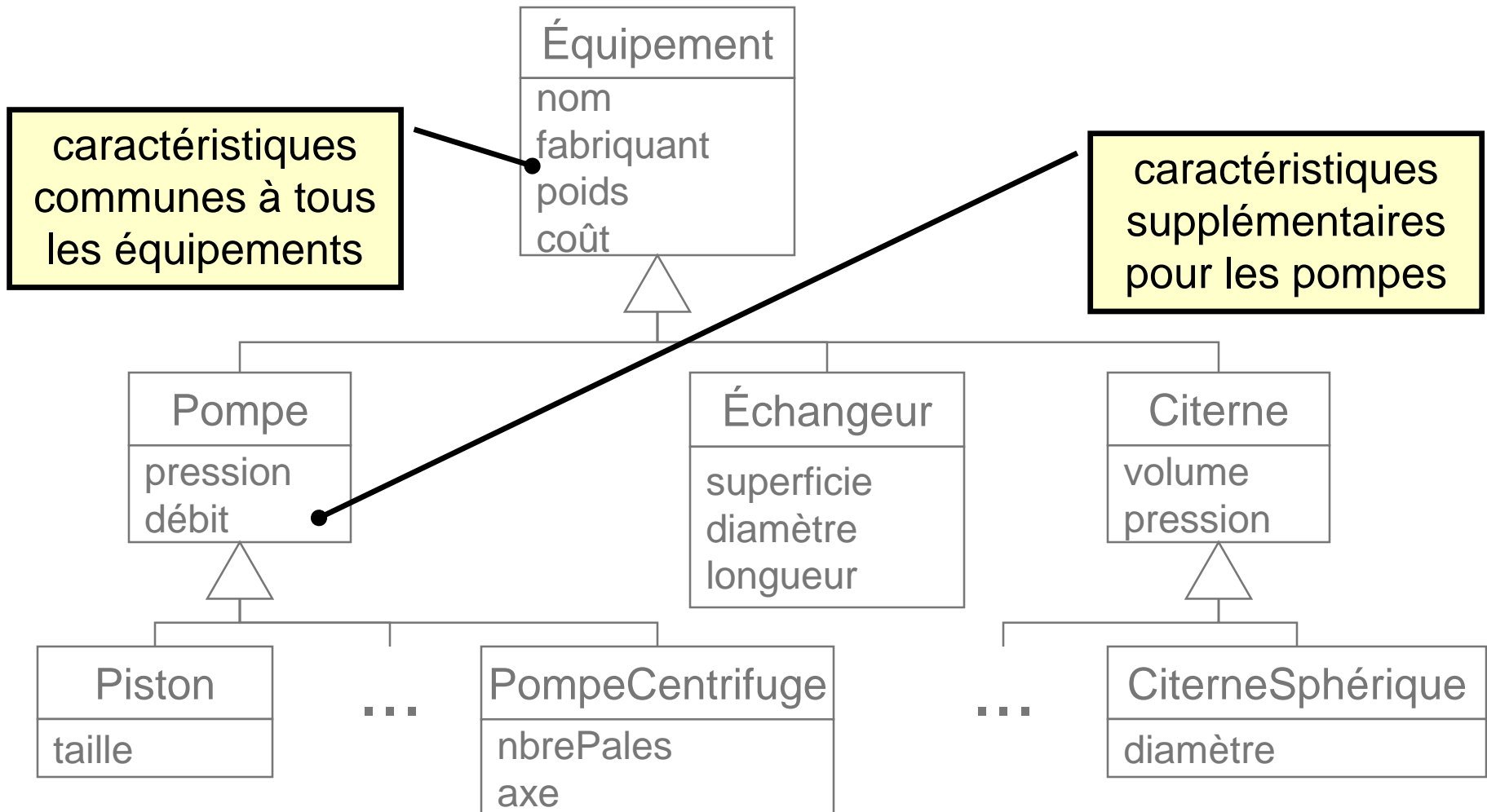
- La spécialisation exprime une relation de “**particularisation**” entre une classe et sa sous-classe (chaque instance de la sous-classe est décrite de manière plus spécifique)

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,



- 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

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 ouns() {  
        System.out.println(«ouns!»);  
    }  
}
```

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

```
C c = new C();
```

```
c.hello();
```

```
c.bye();
```

```
c.ouns();
```

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 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 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) {  
        ...  
    }  
}
```

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 des méthodes

- Annotations¹ (java 1.5) : méta-données 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

```
class A {  
  
    protected double x;  
  
    public void add(double x) {  
        System.out.println("A.add double " + x);  
        this.x += x;  
    }  
}
```

```
class C extends A {  
    Add @Override Annotation  
  
    public void add(double x) {  
        System.out.println("C.add int " + x);  
        this.x += x;  
    }  
}
```

```
class B extends A {  
    method does not override or implement a method from a supertype  
    @Override  
    public void add(int x) {  
        System.out.println("B.add int " + x);  
        this.x += x;  
    }  
}
```

¹ Pour en savoir plus sur les 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

A uneMethode() de C1 est **redéfinie** dans C2
comme B uneMethode()

Redéfinition avec réutilisation

- Redéfinition des méthodes (method overriding) :

- possibilité de réutiliser le code de la méthode héritée (super)*

this permet de faire référence à l'objet en cours

super permet de désigner la superclasse

```
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);  
        ...  
    }  
}
```

l'appel super peut être effectué
n'importe où dans le corps de la méthode

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

- La hiérarchie d'héritage **est un arbre** dont la racine est la classe **Object** (java.lang)

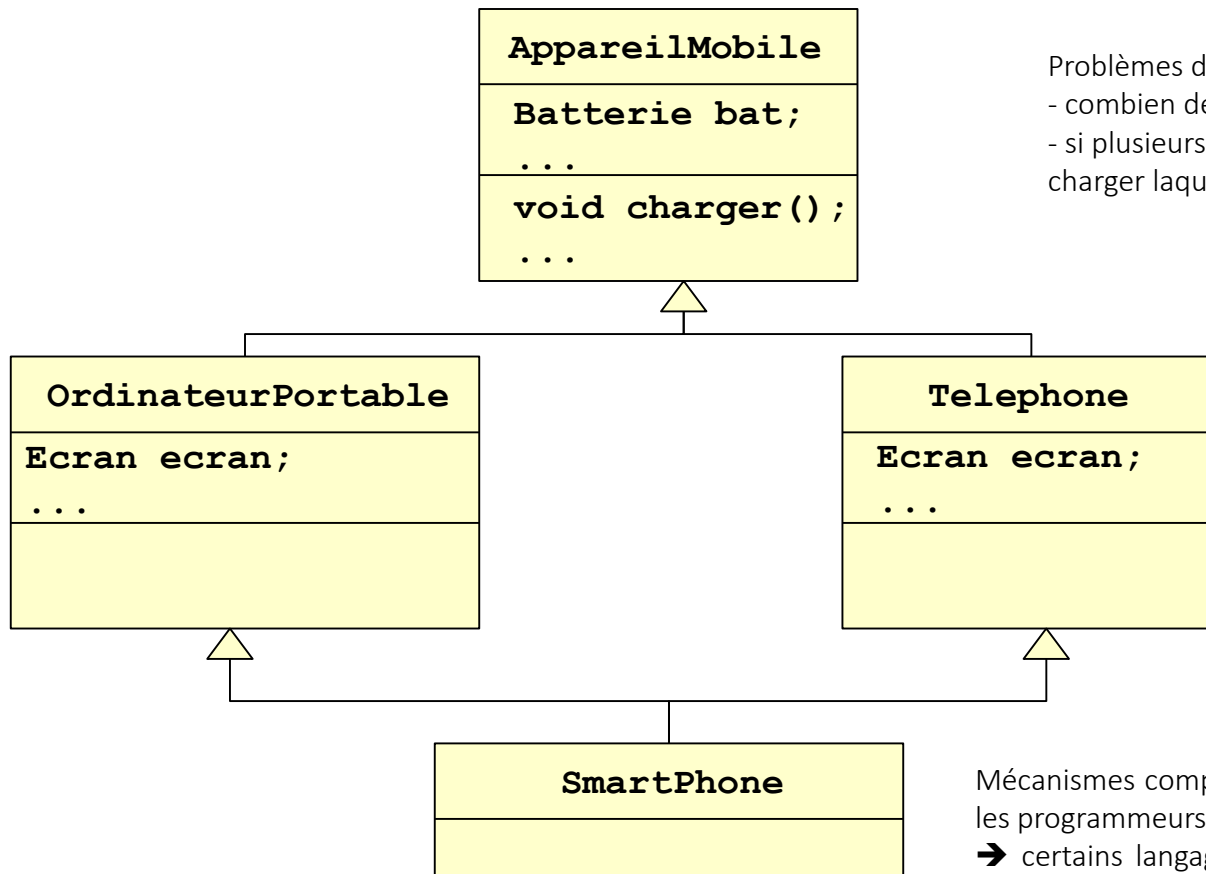
- toute classe autre que **Object** possède une super-classe
- toute classe hérite directement ou indirectement de la classe **Object**
- par défaut une classe qui ne définit pas de clause **extends** hérite de la classe **Object**

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

Problèmes de l'héritage multiple

- Héritage multiple

- une classe peut hériter de plus d'une seule classe



Problèmes de l'héritage en diamant

- combien de batteries possède un smartphone ?
- si plusieurs implémentation 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)

- 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 une 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**, on 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;  
    ...  
}
```

```
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]

Constructeurs

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(...)`*

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 **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***

Héritage

Chaînage des constructeurs

```
public class Object {
```

```
    public Object()
```

```
    {
```

```
        ...
```

```
    }
```

```
    ...
```

```
}
```

```
public class Point extends Object {
```

```
    double x,y;
```

```
    public Point(double x, double y)
```

```
    {
```

```
        super(); // appel implicite
```

```
        this.x = x; this.y = y;
```

```
    }
```

```
    ...
```

```
}
```

```
public class PointCouleur extends Point {
```

```
    Color c;
```

```
    public PointCouleur(double x, double y, Color c)
```

```
    {
```

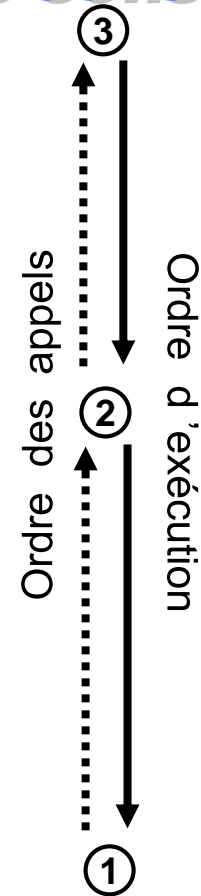
```
        super(x,y);
```

```
        this.c = c;
```

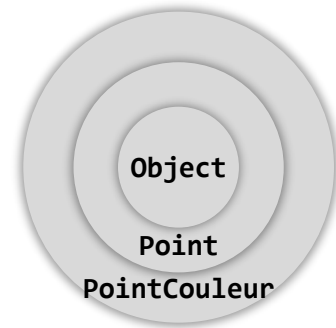
```
    }
```

```
    ...
```

```
}
```



`new PointCouleur(...);`



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

```
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;  
}
```

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

((ClasseA) this) .x

~~**super.super.x**~~

super.x

x ou this.x

Dans le code de ClasseC

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
 - – (visibilité par défaut **package**) 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

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) getX() - 1, (int) getY() - 1, 3, 3);
    }
}
```

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

    ...

    public double getX() {
        return x;
    }

    ...
}
```

Attributs hérités de la
classe Point

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);
    }
}
```

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

    ...

    public double getX() {
        return x;
    }

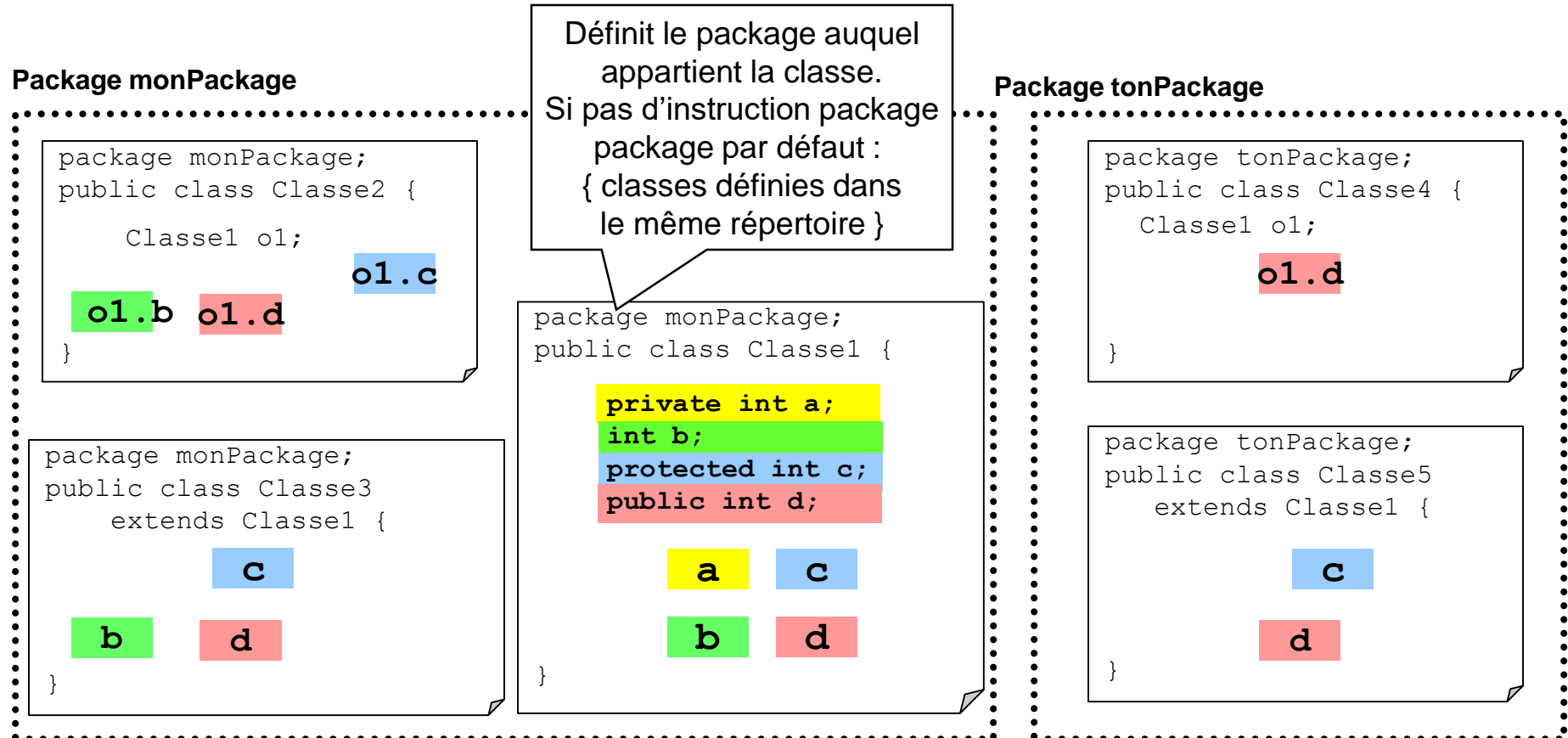
    ...
}
```

Attributs hérités de la
classe Point

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

Héritage

Visibilité des variables et méthodes



Les mêmes règles de visibilité s'appliquent aux méthodes

- 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

Package A

```
package A;  
public class ClasseA {  
  
    ClasseB b;  
  
}
```

```
package A;  
class ClasseB  
    extends ClasseA {  
  
}
```

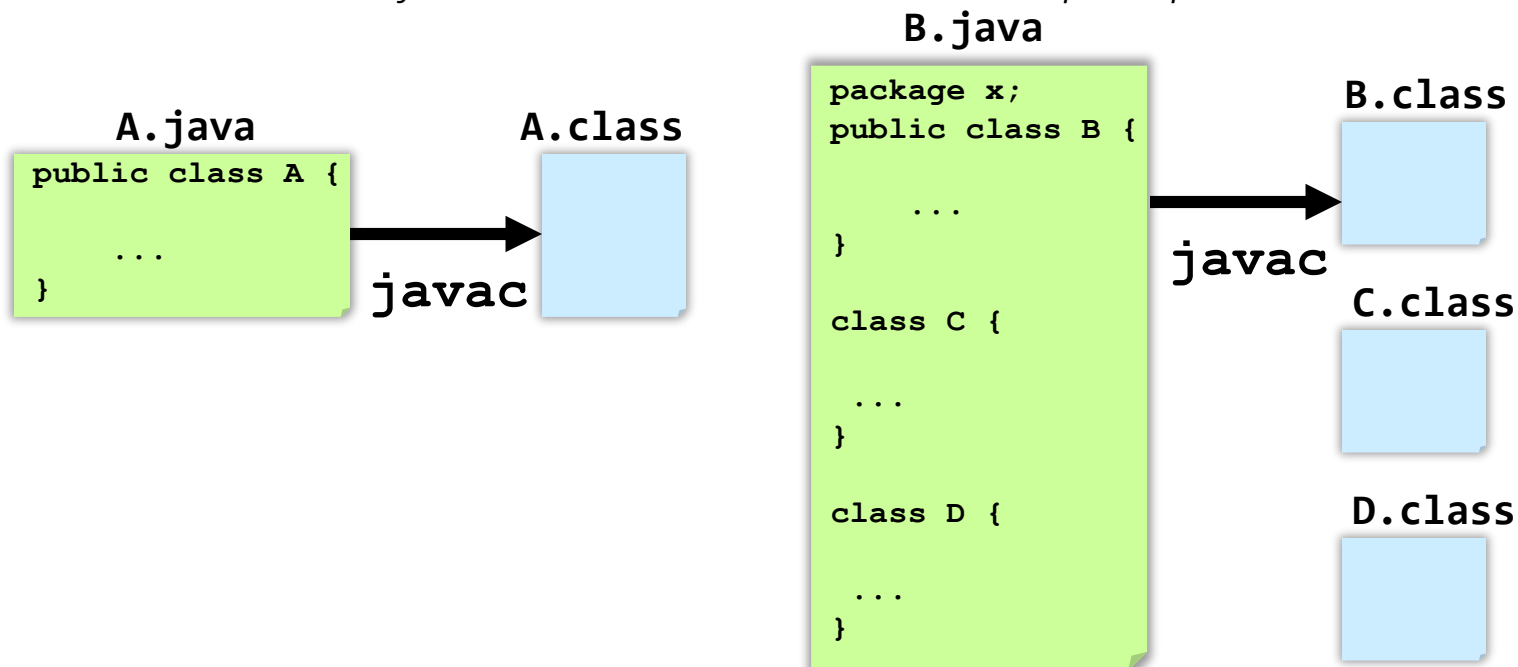
Package B

```
package B;  
import A.ClasseA;  
public class ClasseC {  
  
    ClasseA a;  
    ClasseB b;  
  
}
```

Heritage

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(...)` {
 ... }
- « 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)

Héritage

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 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.

Héritage

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");  
    }  
}
```



??

```
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;  
    }  
}
```

Héritage

Constructeurs et appel de méthodes

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

```
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;  
    }  
}
```

```
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
```

Héritage

Constructeurs et appel de méthodes

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

```
public class ClassB extends ClassA {  
    protected String s;  
    @Override  
    public void foo() {  
        4 super.foo();  
        System.out.println(s + "(length : " + s.length() + ")");  
    }  
    public ClassB(String s) {  
        2 this.s = s;  
    }  
}
```

```
public class ConstructorCallsOverride {  
    public static void main(String[] args) {  
        new ClassB("world"); 1  
    }  
}
```

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

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

5
s n'a pas été initialisée (null)
→ NullPointerException

Héritage

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");  
    }  
}
```

??

```
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;  
    }  
}
```

Héritage

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 = "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

Héritage

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");  
    }  
}
```

```
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;  
    }  
}
```

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

- (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

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

Constructeurs et Méthodes finales

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.

