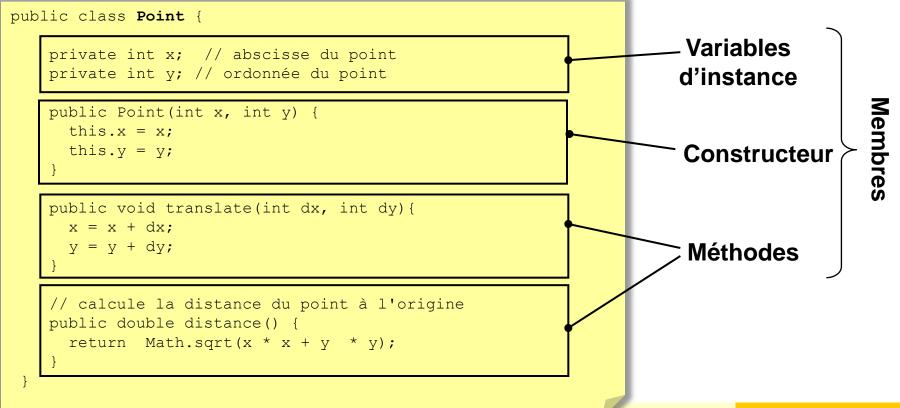
Réutilisation des classes Délégation Héritage

Classe : rappels

- Une classe représente une « famille » d'objets partageant les mêmes propriétés et supportant les mêmes opérations.
- Une classe sert à définir propriétés 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)



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



Mise en œuvre

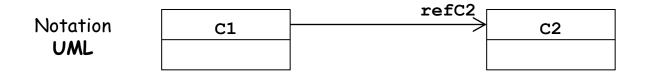
Exemple: la classe Cercle

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

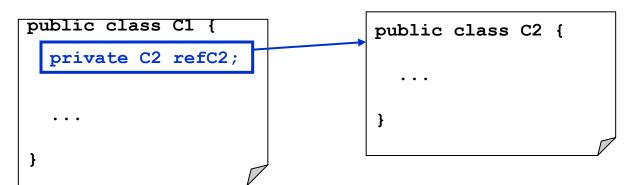


Association

à navigabilité restreinte

La flèche indique que les instances de C1 connaissent les instances de C2 mais pas l'inverse

Traduction en **java**



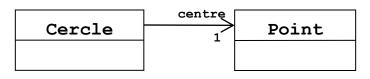
La classe cliente (C1) possède une référence de type de la classe serveuse (C2)

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);
```

UJF

Délégation : exemple

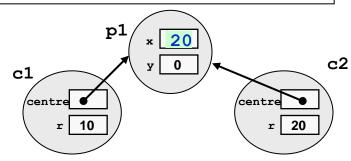
```
public class Cercle {
 /** centre du cercle */
  private Point centre;
 /** rayon du cercle*/
  private double r;
  public Cercle( Point centre, double r) {
      this.centre = centre;
      this.r = r:
  public void translater(double dx, double dy) {
    centre.translater(dx,dy);
                 c2.translater(10,0);

    Affecte aussi cercle c1

           Après ces opérations le centre des 2 cercles est (20,0)
                   p1.rotation(90);
```

- 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'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)

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;
   }
   ...
}
```

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

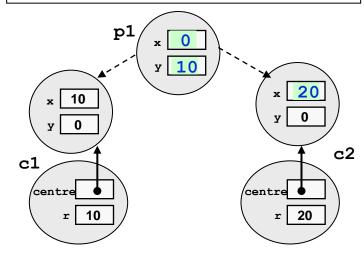
N'affecte que le cercle c2

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);
...
```

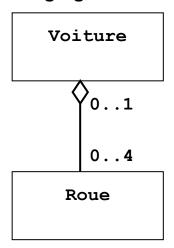


 les cycles de vies du Point et et du Cercle sont liés : si le cercle est détruit (ou copié), le centre l'est aussi.

Agrégation / Composition

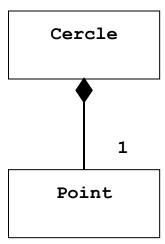
- Les deux exemples précédents traduisent deux nuances (sémantiques) de l'association a-un entre la classe Cercle et la classe Point
- UML distingue ces deux sémantiques en définissant deux type de relations :

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

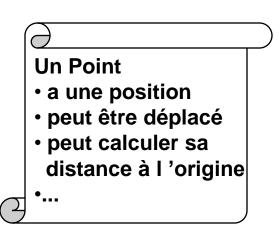
- Extension d'une classe
- Terminologie
- Généralisation/spécialisation
- Héritage en Java
- Redéfinition des méthodes
- Réutilisation
- Chaînage des constructeurs
- Visibilité des variables et des méthodes
- Classes et méthodes finales



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 (inheritence)
 - définir une nouvelle classe à partir de la classe déjà existante





Syntaxe java

• 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

UJF



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

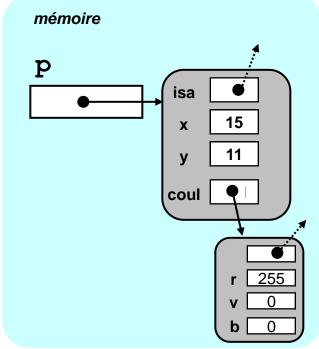
```
PointGraphique p = new PointGraphique();

// 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);

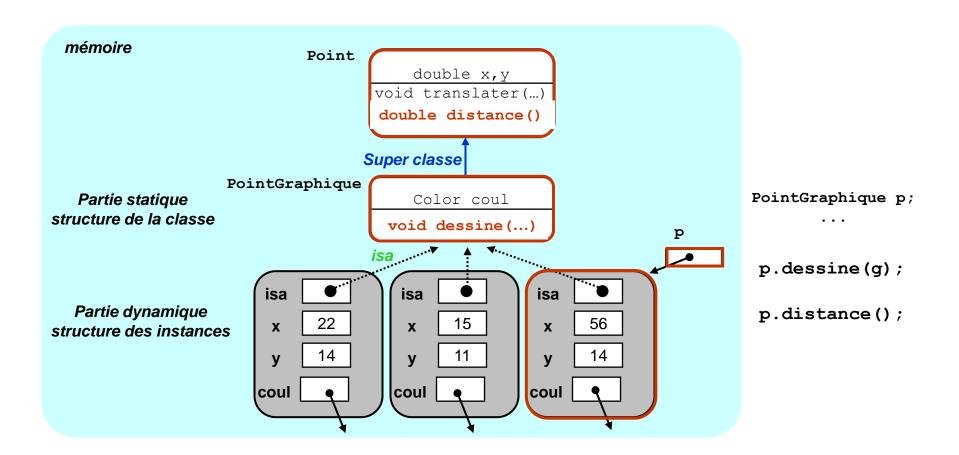
// utilisation d'une méthode héritée
double dist = p.distance();

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





résolution des messages





Terminologie

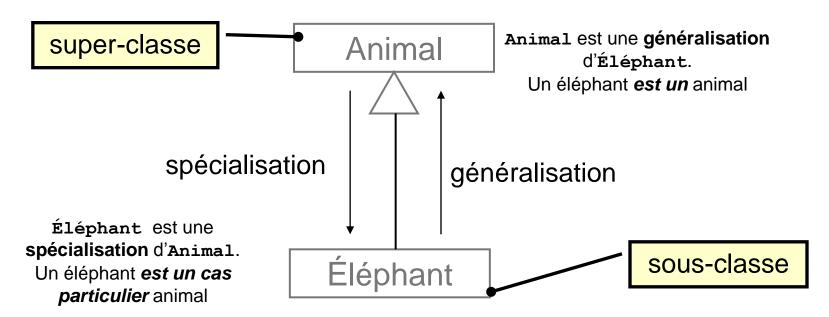
- **Héritage** permet de reprendre les caractéristiques d'une classe **m** existante pour les étendre et définir ainsi une nouvelle classe **F** qui hérite de **m**.
- Les objets de F possèdent toutes les caractéristiques de M avec en en plus celles définies dans F
 - Point est la classe mère et PointGraphique 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 superclasse (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)

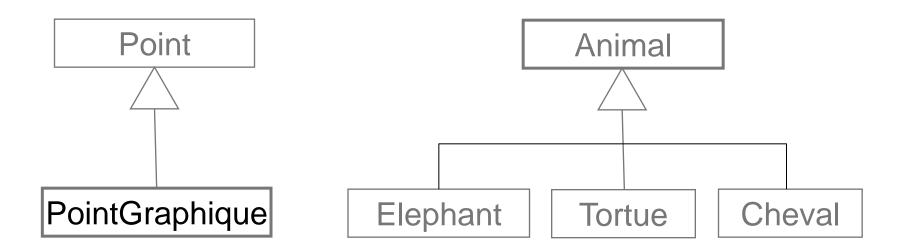


© Philippe GENOUD



Généralisation/Spécialisation

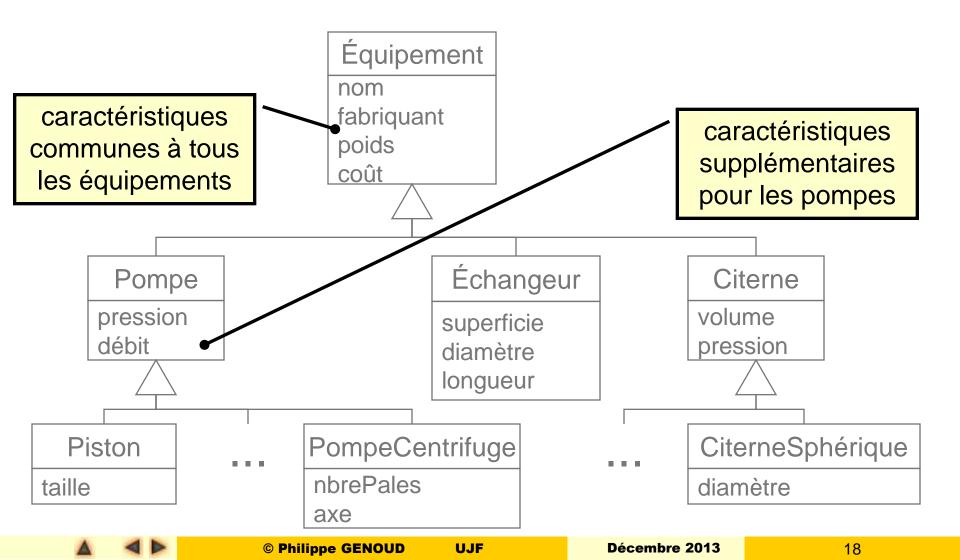
- Utilisation de l'héritage :
 - dans le sens "spécialisation" pour réutiliser par modification incrémentielle les descriptions existantes.
 - dans le sens "généralisation" pour abstraire en factorisant les propriétés communes aux sous-classes,







- 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

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

```
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!»);
   }
}
```

```
C c = new C();
c.hello();
c.bye();
c.oups();
```

UJF



- 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.

UJF





```
public class A {

public void hello() {
    System.out.println(«Hello»);
}

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

```
public class B extends A {
  public void affiche() {
    System.out.println(«Je suis un B»);
  }
}
```

```
A a = new A();
B b = new B();
a.hello(); --> Hello
a.affiche(); --> Je suis un A
b.hello(); --> Hello
b.affiche(); --> Je suis un B
```



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

B possède deux méthodes methodX (methodX(int) et methodX(Color)) C possède une seule méthode methodX (methodX(int))



- Annotations¹ (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

```
class A {
    protected double x;

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

¹Pour en savoir plus sur les annotations :

http://adiguba.developpez.com/tutoriels/java/tiger/annotations/





- 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



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 public class Etudiant { **super** permet de désigner la superclasse 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; l'appel super peut être effectué public void affiche() n'importe où dans le corps de la super.affiche(); méthode System.out.println("Sport pratiqué : "+sportPratiqué);



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
```

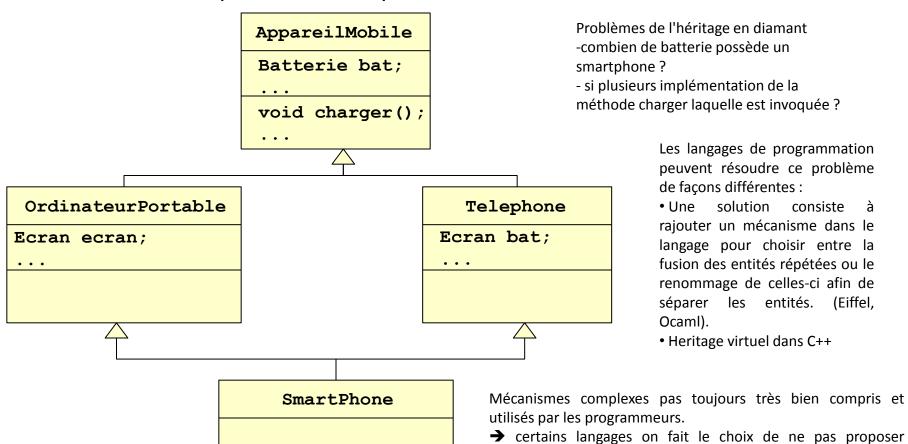
UJF



Problèmes de l'héritage multiple

Héritage multiple

une classe peut hériter de plus d'une seule classe



l'héritage multiple (Java, c##, Ruby, ObjectiveC)



La classe Object

- Principales méthodes de la classe Object
 - public final <u>Class</u> 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 <u>Object</u> 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)

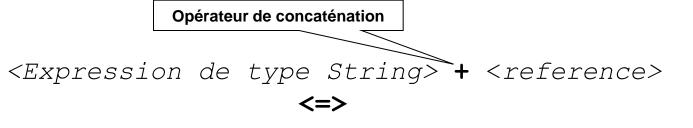
public <u>String</u> toString()

Renvoie une chaîne représentant la valeur de l'objet

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



A propos de toString



<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() + "@" +
                                      public class Object {
     Integer.toHexString(hashCode());
                                      public class Point {
                                                                  public String toString(){
                                                                     return "Point:[" + x +
                                        private double x;
                                                                                  "," + v + "]");
                                        private double y;
        La classe Point ne
                                                                           La classe Point
       redéfinit pas toString
                                                                           redéfinit toString
                                 Point p = new Point(15,11);
                                                                        Point: [15.0,11.0]
         Point@2a340e
                                    System.out.println(p);
```

UJF

29



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)
```

UJF

```
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

UJF





```
Chaînage des constructeurs
public class Object {
  → public Object()
                                                                 appels
public class Point extends Object {
    double x, y;
    public Point(double x, double y)
                                                                         d 'exécution
                                                                    2
                                                                 des
          super(); // appel implicite
                                                                 Ordre
          \overline{\text{this.x}} = x; \text{ this.y} = y;
public class PointCouleur extends Point {
    Color c;
    public PointCouleur(double x, double y, Color c)
           super(x,y);
                                                            new PointCouleur(...);
          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;
                                            public A() {
                                                               Constructeur
                                                                 par défaut
                                               super();
    // méthodes
                                                                 implicite
    String getNom() {
      return nom;
                                                             Garantit chaînage
                                                             des constructeurs
```



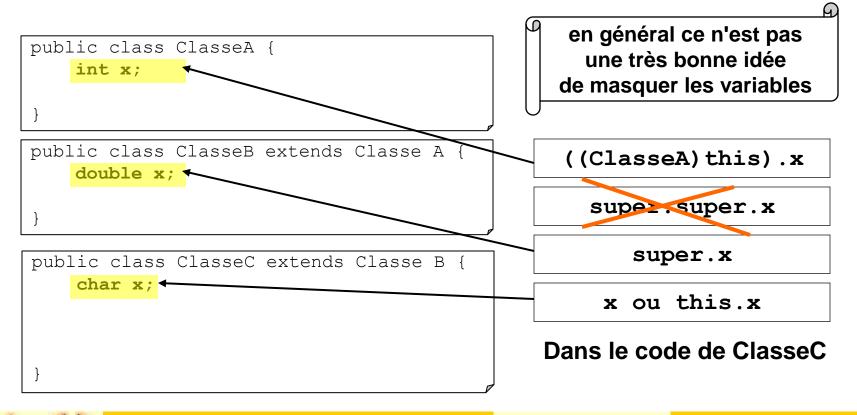
Constructeur par défaut

```
public class ClasseA {
    double x;
    // constructeur
                                               Constructeur explicite
    public ClasseA(double x)
                                           masque constructeur par défaut
            this.x = x;
                                                      Pas de constructeur
                                                       sans paramètres
public class ClasseB extends ClasseA {
    double y = 0;
                                             public ClasseB() { | Constructeur
                                                                  par défaut
                                               super();
    // pas de constructeur
                                                                  implicite
```



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





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





Syntaxe java

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.setColor(coul);
      getX()
                             gety()
```

```
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



Syntaxe java

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



Visibilité des variables et méthodes

Package monPackage

```
Définit le package auquel
appartient la classe.
Si pas d'instruction package
package par défaut :
{ classes définies dans
le même répertoire }
```

Package tonPackage

```
package tonPackage;
public class Classe4 {
  Classel ol;
         o1.d
package tonPackage;
public class Classe5
   extends Classel {
             C
```

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

UJF

Package A

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

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

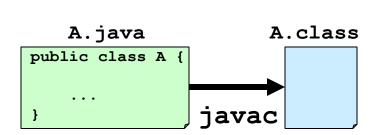
Package B

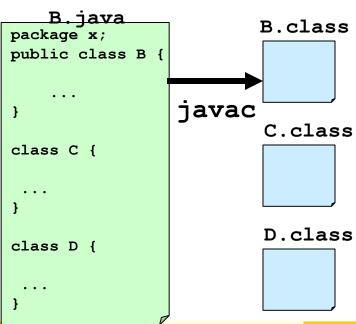
```
package B;
public class ClasseC {
   ClasseA a;
   Class B b;
}
```



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

- 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)

UJF



44



- 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.

UJF