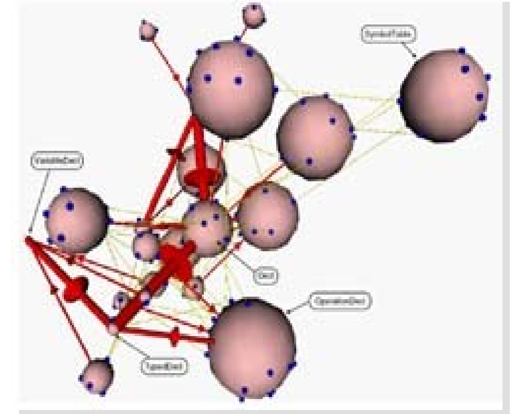
La programmation Orientée Objets et le langage Java

Belgacem BEN HEDIA
ESISAR-INP Grenoble

CITI - INSA Lyon
EPT

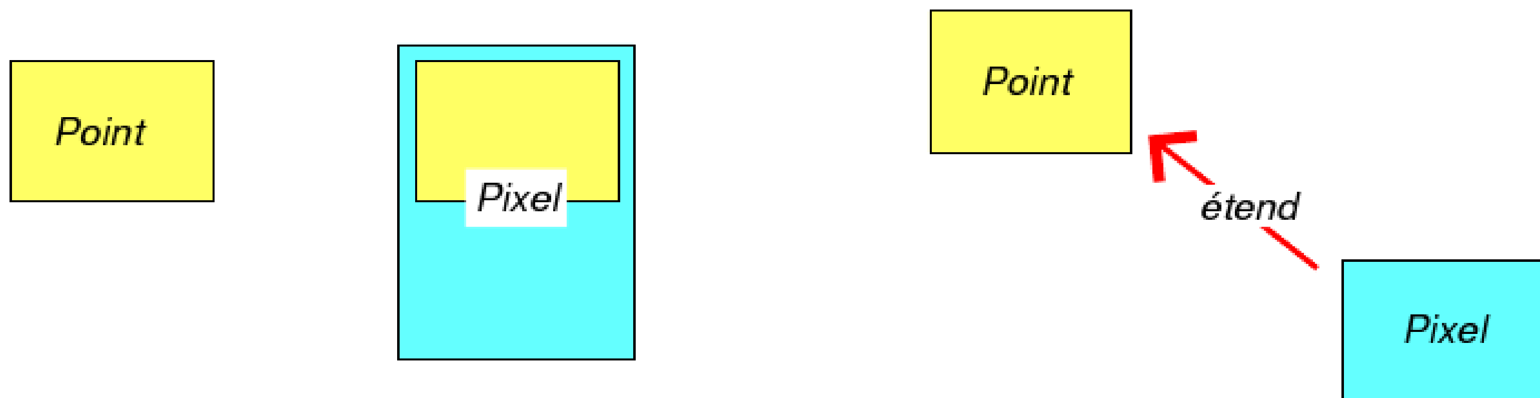


Cours 2: « L'héritage »

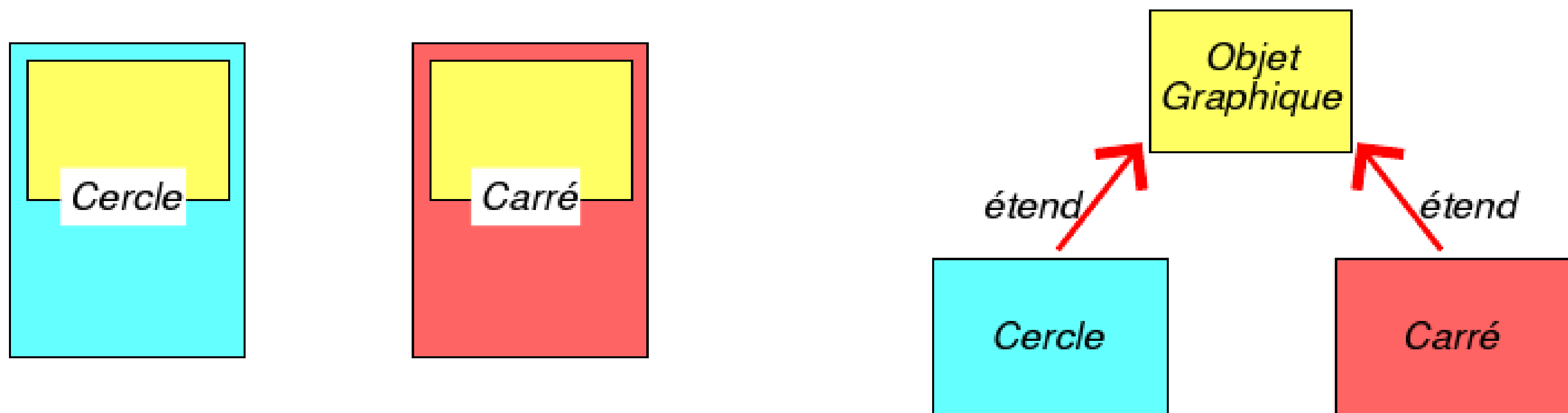
belgacem.ben-hedia@insa-lyon.fr
belgacem.ben-hedia@esisar.inpg.fr
http://www.bbhedia.org

Héritage : la problèmatique

- Une classe est extension d'une autre (*réutilisation*)



- Plusieurs classes ont une partie commune (*abstraction*)



Héritage

- mécanisme pour définir une nouvelle classe comme extension d'une classe préexistante
- tous les membres de la classe préexistante sont membres de la nouvelle classe

Fichier Point.java

```
public class Point {
   int x, y ;
   void placer(int a, int b) {
       x = a ;
       y = b ;
   }
   ...
}
```

Fichier Pixel.java

```
public class Pixel extends Point {
       Color couleur ;
       void colorier(Color c) {
           couleur = c ;
       }
       ...
}
```

un objet `Point`

10	x
20	y

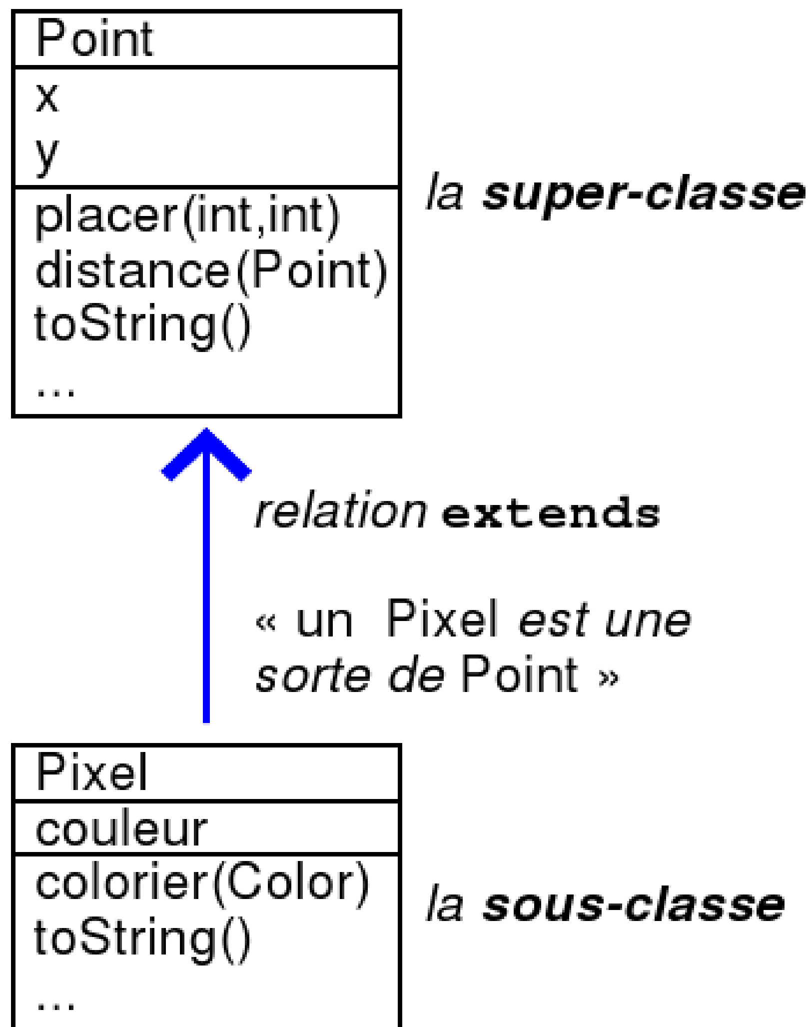
un objet `Pixel`

10	x
20	y
red	couleur

```
Pixel pix = new Pixel() ;

pix.placer(10, 20) ;
pix.colorier(Color.red) ;
```

Héritage, un peu de jargon



Pas de contrainte-surprise :

la super-classe n'a pas à être

- ni dans le même package
- ni dans le même dossier
- ni disponible sous forme de fichier source

Héritage...

- de l'état : les objets de la sous-classe ont tous les champs de la super-classe
- du comportement : tout ce qu'un objet de la super-classe sait faire, un objet de la sous-classe sait le faire aussi.

Fichier Point.java

```
class Point {
   int x, y ;
   double distance(Point p) {
       int dx = x - p.x ;
       int dy = y - p.y ;
       return Math.sqrt(dx * dx + dy * dy) ;
   }
   ...
}
```

Fichier Pixel.java

```
class Pixel extends Point {
       Color couleur ;
       ...
}
```

exemple, avec

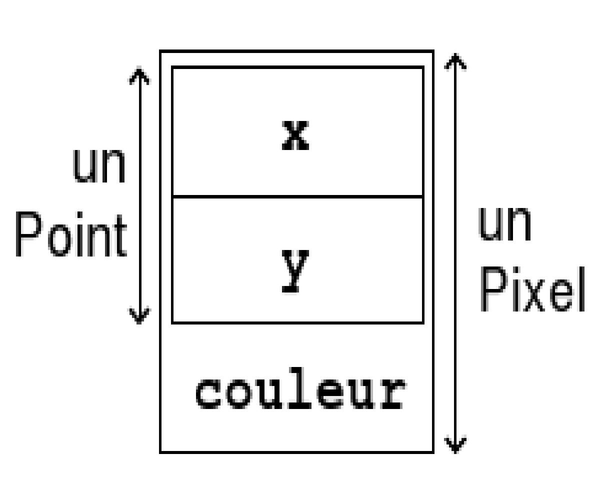
```
Point point1, point2 ;
Pixel pixel1, pixel2 ;
```

initialisation de ces variables

ces expressions sont légitimes :

```
point1.distance(point2)
pixel1.distance(point2)
point1.distance(pixel2)
pixel1.distance(pixel2)
```

Héritage et constructeur



- la construction d'une instance de la sous-classe *commence* par la construction de sa partie héritée
- en clair : qu'on le veuille ou non, pour initialiser un `Pixel` il faut commencer par l'initialiser en tant que `Point`

- si on ne fait rien, `javac` insère au début de chaque constructeur de la sous-classe un appel du constructeur sans argument de la super-classe
- problème : et si un tel constructeur n'existe pas ?

Héritage et constructeur

- Constructeur problématique

Fichier Pixet.java *erroné*

```
class Pixel extends Point {
      Color couleur ;
      Pixel(int a, int b, Color c) {
          ici se cache un appel implicite de Point()
          x = a ;
          y = b ;
          couleur = c ;
      }
      ...
}
```

- le constructeur Point() existe-t-il ?
- si `Point()` existe, `x` et `y` sont-ils accessibles dans `Pixel` ? (probablement non)
- même si `Point()` existe et `x` et `y` sont accessibles, il est maladroit de les initialiser pour rien, puisque tout de suite apr`es ont leur affecte d'autres valeurs

Héritage et constructeur

- La solution

Fichier Pixet.java

```
class Pixel extends Point {
      Color couleur ;
      Pixel(int a, int b, Color c) {
          super(a,b)
          couleur = c ;
      }
      ...
}
```

- cela se lit : « pour initialiser un `Pixel` à partir de `a, b` et `c`, commencez par l'initialiser en tant que `Point` à partir de `a` et `b`, ensuite donnez à `couleur` la valeur `c` »
- `super(...)` doit être la première instruction d'un constructeur

Héritage et droit d'accés

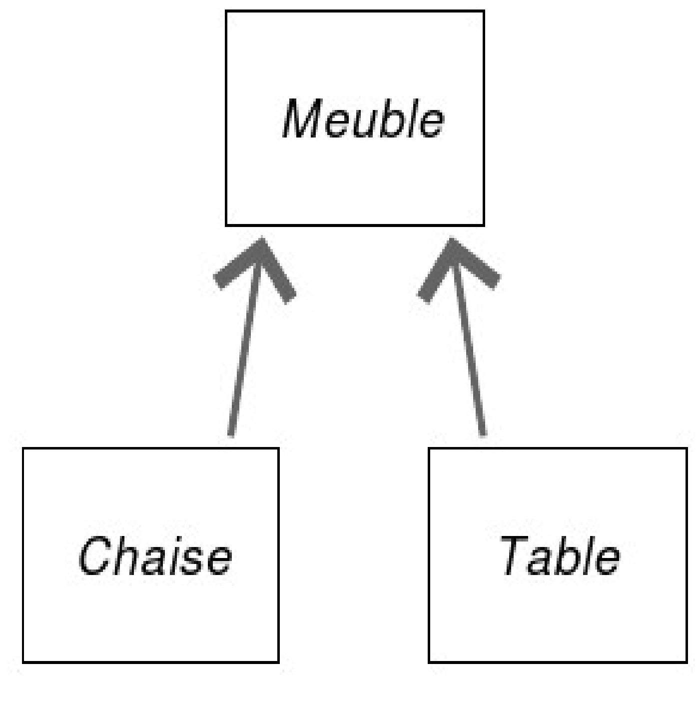
- `protected` : permission intermédiaire entre `private` et `public`
- un membre `protected` est accessible dans les sous-classes, mais pas partout
- idée : l'auteur d'une sous-classe a plus de droits que le commun des mortels
- exemple : le constructeur protégé d'une classe « abstraite »

Fichier Meuble.java

```
class Meuble {
        protected Meuble(...) {
            ...
        }
        ...
}
```

Fichier Chaise.java

```
class Chaise extends Meuble {
        public Chaise(...) {
            super(...) ;
            ...
        }
        ...
}
```



Surcharge et redéfinition des noms des membres

- héritage : que se passe-t-il si un membre de la super-classe a le même nom qu'un membre de la sous-classe ?
- s'il s'agit d'une variable et une méthode, ou de deux méthodes de signatures différentes :
  - ils coexistent (mécanisme de la surcharge)
- s'il s'agit de deux variables ou de deux méthodes de même signature :
  - le membre de la sous-classe masque celui de la super-classe
  - s'il s'agit de deux variables, c'est généralement une maladresse
  - cas particulier de deux méthodes de même signature : on appelle cela une redéfinition de méthode et c'est extrêmement utile

Masquage de variables membres

- masquer n'est pas rendre inaccessible, *pour les variables*
- exemple (peu utile !) :

```
class Article {
    int code = 111 ;
    ...
}
class Alimentation extends Article {
    int code = 222 ;
    ...
}
class RayonFrais extends Alimentation {
    int code = 333 ;
    ...
    void test() {
        System.out.println( code ) ;                      // ceci écrit 333
        System.out.println( super.code ) ;                // ceci écrit 222
        System.out.println( ((Article) this).code ) ;     // ceci écrit 111
    }
}
```

- autant que possible, éviter de telles situations si malcommodes (les variables devraient avoir des noms plus significatifs !)

Cas des méthodes : la surcharge

```
class Point {
    ...
    void deplacer(int dx, int dy) {
        changement de la position du point
    }
}
class Pixel extends Point {
    ...
    void deplacer(Color k) {
        changement de la couleur du pixel
    }
}
Pixel pix = new Pixel(...) ;
...
pix.deplacer(u, v) ;                appel de deplacer heritée de Point
...
pix.deplacer(w) ;                   appel de deplacer définie dans Pixel
```

Cas des méthodes : la redéfinition

- si une méthode de la sous-classe a la même signature (nom et arguments) qu'une méthode de la super-classe : on dit que la méthode est *redéfinie* dans la sous-classe
- justification : puisque la sous-classe *raffine* la super-classe, c'est normal que certaines méthodes de la super-classe y aient une nouvelle version
- autrement dit, les objets de la sous-classe savent faire tout ce que savent faire les
- objets de la super-classe, mais certaines choses « mieux »
- exemples (très basiques) : les méthodes `toString`, `equals`, `clone`, etc.

Cas des méthodes : la redéfinition

Fichier Point.java

```
class Point {
    private int x, y ;
    ...
    public String toString() {
        return "(" + x + "," + y + ")" ;
    }
    ...
}
```

Un point sous forme de chaîne :

`"(10,20)"`

Un pixel sous forme de chaîne :

`"(10,20)-red"`

Fichier Pixel.java

```
class Pixel extends Point {
   private Color couleur ;
   ...
   public String toString() {
       return "(" + x + "," + y + ")-" + couleur ;
   }
   ...
}
```

Erreur :

`x` et `y` sont prives.

Cas des méthodes : la redéfinition

Fichier Point.java

```
class Point {
    protected int x, y ;
    ...
    public String toString() {
        return "(" + x + "," + y + ")" ;
    }
    ...
}
```

Un point sous forme de chaîne :

`"(10,20)"`

Un pixel sous forme de chaîne :

`"(10,20)-red"`

Fichier Pixel.java

```
class Pixel extends Point {
   private Color couleur ;
   ...
   public String toString() {
       return "(" + x + "," + y + ")-" + couleur ;
   }
   ...
}
```

Cela passe, mais c'est mal conçu : la classe `Pixel` s'appuie sur des détails internes de classe `Point`

Cas des méthodes : la redéfinition

Fichier Point.java

```
class Point {
    private int x, y ;
    ...
    public String toString() {
        return "(" + x + "," + y + ")" ;
    }
    ...
}
```

Un point sous forme de chaîne :

```
"(10,20)"
```

Un pixel sous forme de chaîne :

```
"(10,20)-red"
```

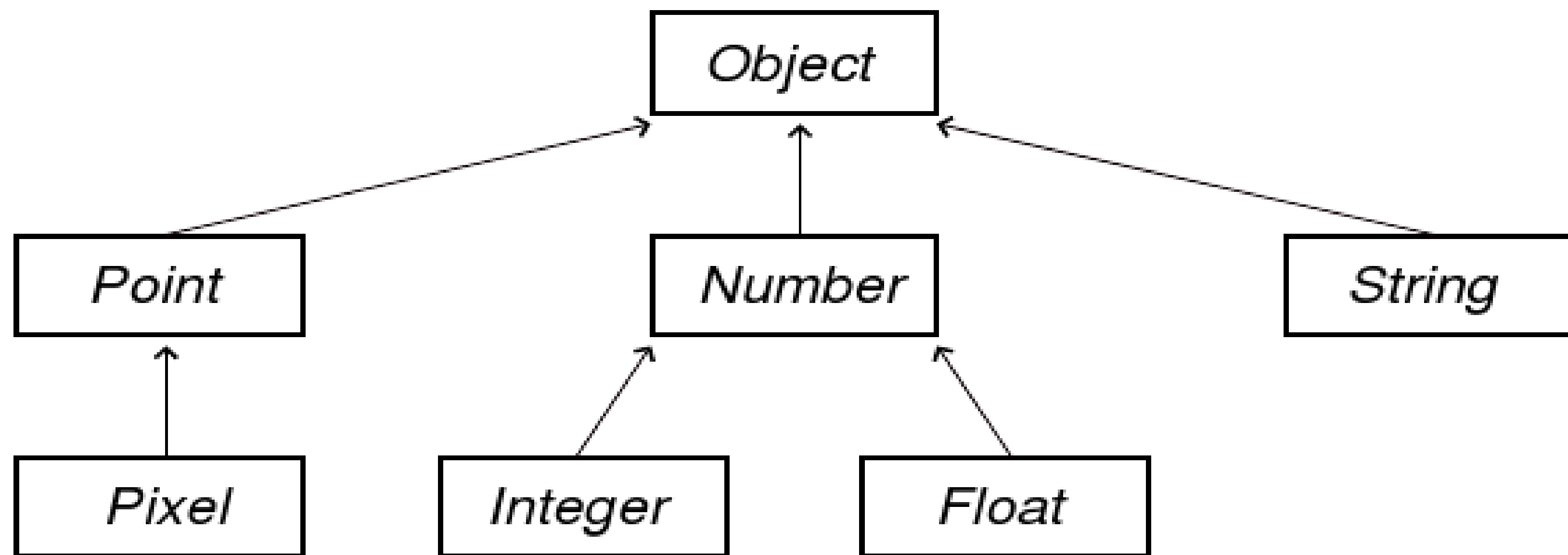
Fichier Pixel.java

```
class Pixel extends Point {
   private Color couleur ;
   ...
   public String toString() {
       return super.toString + "-" + couleur ;
   }
   ...
}
```

La bonne solution

L'arbre de toutes les classes

- Certaines classes ont une super-classe explicite
- Il existe une classe, nommée `Object`, qui n'a pas de super-classe
- Toutes les autres classes sont sous-classes de `Object`
- L'ensemble des classes est organise en une arborescence de racine `Object`



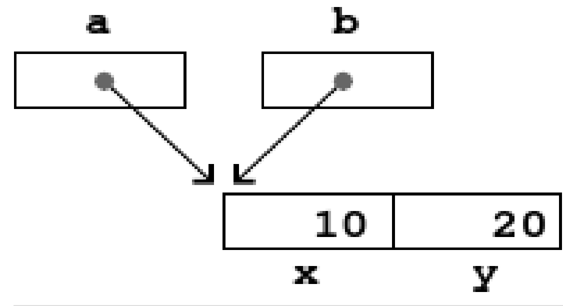
- N.B. Les classes sont nommées par le substantif d'une instance

(L'affectation et la comparaison des objets)

- l'affectation « `a = b` » d'un objet n'en fait pas une copie

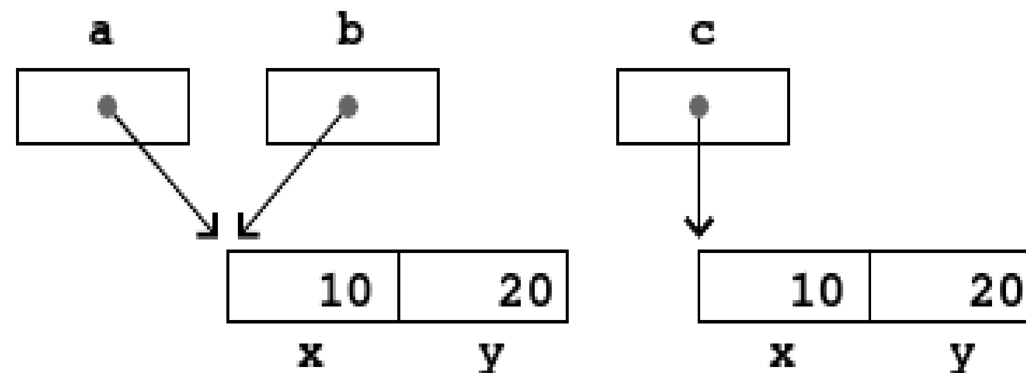
```
Point a = new Point(10, 20) ;
Point b = a ;
```

- `a` et `b` ne sont pas les noms de deux objets, mais deux noms pour le même objet :



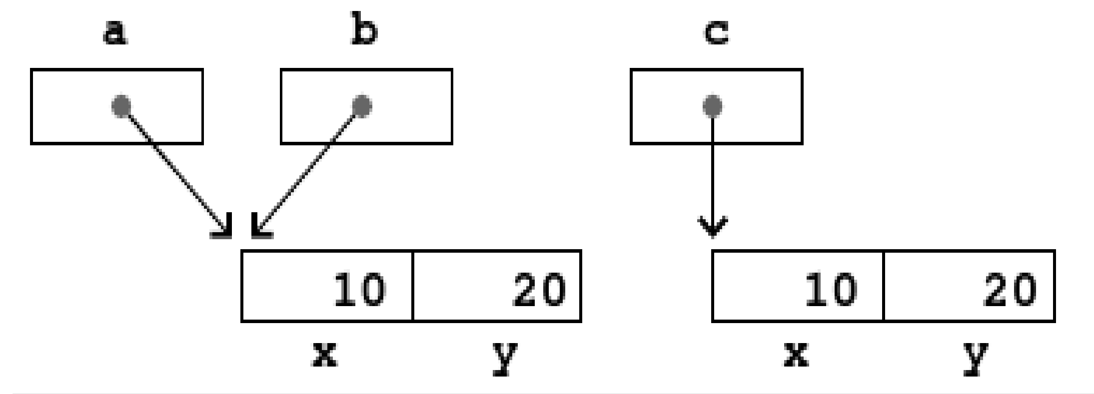
- pour avoir une vraie duplication :

```
Point c = (Point) a.clone() ;
```



(L'affectation et la comparaison des objets)

- de même, « `a == b` » ne signifie pas « `a est egal à b` », mais « `a et b sont le même objet` » :



`a == b` *vaut* `true`

`a == c` *vaut* `false`

- pour avoir une comparaison des valeurs :

`a.equals(b)` *vaut* `true`

`a.equals(c)` *vaut* `true` *// si la classe Point a été bien programmée*

Qu'y a-t-il dans la classe `Object` ?

Réponse évidente : le comportement commun à tous les objets :

`toString()` expression de l'objet sous forme de chaîne de caractères

par défaut : *`classe@adresse ;`* exemple : *`Point@18d107f`*

`clone()` renvoie un clone (duplicata) de l'objet

par défaut : copie « à un niveau »

(duplication de l'objet mais non de ses membres)

`equals()` comparaison *des valeurs* de deux objets :

`a.equals(b)` ⇔ la valeur de `a` est égale à celle de `b`

par défaut, `a.equals(b)` ⇔ `a == b`

`getClass()` renvoie la classe de l'objet

`a.getClass() == b.getClass()` ⇔ `a` et `b` sont de même type

Opérateur `instanceof`

`unObjet instanceof uneClasse`

- signifie : *`unObjet` est-il instance de [une sous-classe de] `uneClasse` ?*
- c'est-à-dire : *`unObjet` est-il une sorte de `uneClasse` ?*
- application : redéfinition de la méthode `equals`

Fichier Point.java

```
class Point {
    private int x, y ;
    ...
    public boolean equals(Point p) {
        return p.x == x && p.y == y ;
    }
    ...
}
```

Erreur ! Cette méthode est correcte, mais elle n'est pas une redéfinition de `equals(Object p)`

Opérateur `instanceof`

`unObjet instanceof uneClasse`

- signifie : `unObjet` *est-il instance de [une sous-classe de]* `uneClasse` *?*
- c'est-à-dire : `unObjet` *est-il une sorte de* `uneClasse` *?*
- application : redéfinition de la méthode `equals`

Fichier Point.java

```
class Point {
    private int x, y ;
    ...
    public boolean equals(Point p) {
        return p instanceof Point &&
            ((Point)p).x == x && ((Point)p).y == y ;
    }
    ...
}
```

La première condition justifie les deux changements de type

Classes-enveloppes des types primitifs

- les types primitifs ne sont pas dans l'arbre des classes
- de nombreux outils de la bibliothèque ne sont définis que pour des objets
- on convertit en objets les valeurs des huit types primitifs à l'aide des huit

classes-enveloppes :

```
Byte        Float
Short       Double
Integer     Character
Long        Boolean
```

- principaux membres (par exemple, pour `Integer`)

« emballage » : `unInteger = new Integer(unInt) ;`

ou : `unInteger = Integer.valueOf(i)`

« deballage » : `unInt = unInteger.intValue() ;`

utilitaires : `unInt = Integer.parseIntg(uneChaıne) ;`

- N.B. En Java 5 cette question est bien améliorée

Polymorphisme

- notion qui apparaît à plusieurs endroits de la POO
- ici : polymorphisme = un objet peut apparaître sous divers types
- pour commencer, la généralisation : un objet peut apparaître sous un type plus général que le sien :
  - la classe `Pixel` est sous-classe de `Point`, donc
  - tous les membres de `Point` sont membres de `Pixel` et tout ce qu'on peut demander *à un `Point` on peut le demander à un `Pixel`*, donc
  - là ou un `Point` est attendu, on peut mettre un `Pixel`
  - aussi longtemps que cela durera, le `Pixel` sera vu comme un `Point`

Généralisation

une méthode « attendant » un point

```
static void unTraitement(Point pt) {
    ...
    la valeur de pt est un Point
    ...
    pt.placer(a, b) ;
    ...
}
```

Il vaudrait mieux dire : *la valeur de* `pt` *est « au moins » un* `Point`

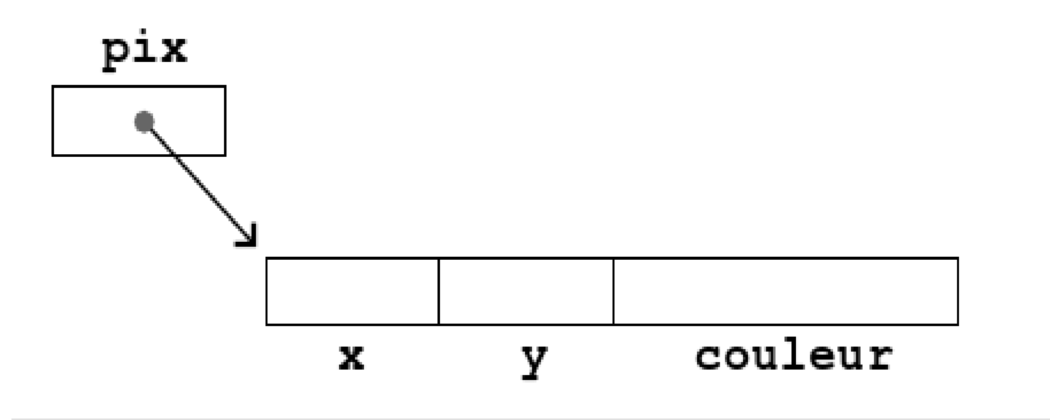
un appel de cette méthode

```
...
Pixel pix = new Pixel(10, 20, pink) ;
...
unTraitement(pix) ;
...
```

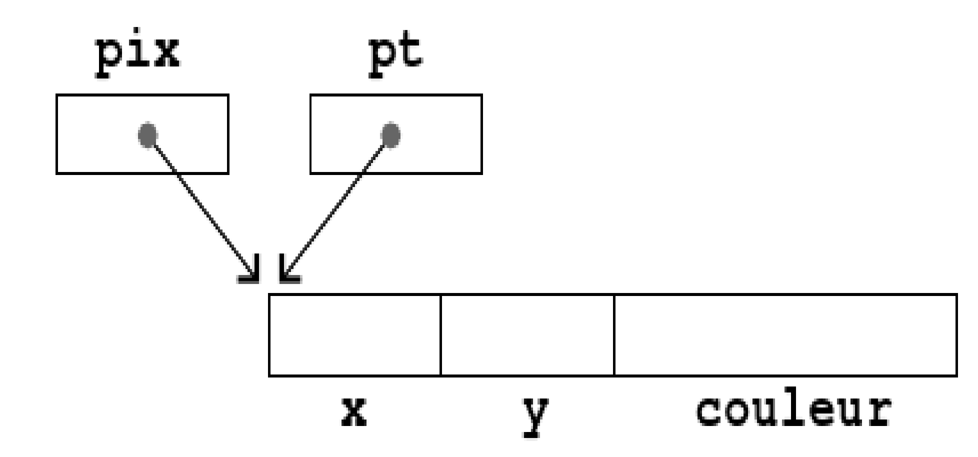
Généralisation

La généralisation est une opération *sans travail* :

```
Pixel pix = new Pixel(a, b, Color.green) ;
...
```




```
Point pt = pix ;
```



Particularisation

- opération consistant à voir un objet sous un type plus précis (moins général) que celui sous lequel il apparaît à un moment donne
- alors que la généralisation est toujours *légitime* et *implicite*,
- la particularisation
  - doit être explicite (par un changement de type)
  - est sous la responsabilité du programmeur
  - toujours contrôlée par Java, à la compilation et à l'exécution

exemple bien glauque

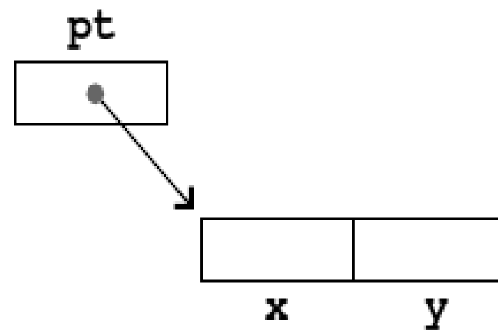
```
static void unTraitement(Point pt) {
    ...
    Pixel pix = (Pixel) pt ;
    ...
    pix.couleur = Color.black ;
    ...
}
```

Sera contrôlé par Java durant l'exécution

Particularisation

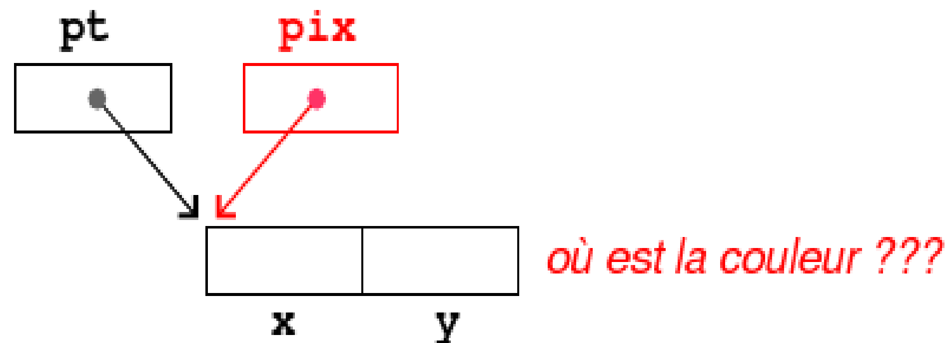
- Comme la généralisation, la particularisation est une opération *sans travail*, mais plus « risquée »
- Elle ne peut servir qu'à donner à un objet un type *qu'il* a déjà

```
Point pt = new Point(a, b) ;
...
```




```
Point pt = new Point(a, b) ;
```
ERREUR !

```
...
```



Exemple : les collections

- Les collections (`classes Collection, List, Vector, Set, Map, etc.`) de la bibliothèque `Java` sont définies en toute généraliste
- leurs éléments sont donc déclarés comme des `Object`
- exemple : la classe `Stack` (pile) :

```
public class Stack extends Vector {
  boolean empty() ;
     /* Teste si la pile est vide */
  void push(Object item) ;
     /* Empile l'objet indique */
  Object pop() ;
     /* Enlève l'objet au sommet de la pile et le renvoie*/
}
```

Exemple : les collections

- Exemple d'utilisation de la pile : un programme crée des points, les exploite une première fois en les empilant, puis une deuxième fois en les dépilant
- déclaration et initialisation de la pile

```
Stack pile = new Stack() ;
```

- empilement des points

```
for (int i = 0 ; i < n ; i++) {
   Point pt = new Point(..., ...) ;
   exploitation de pt
   pile.push(pt) ;        (généralisation)
}
```

- dépilement des points

```
while ( !pile.empty()) {
   Point pt = (Point) pile.pop() ;  (particularisation)
   exploitation de pt
}
```

Polymorphisme et méthodes virtuelles

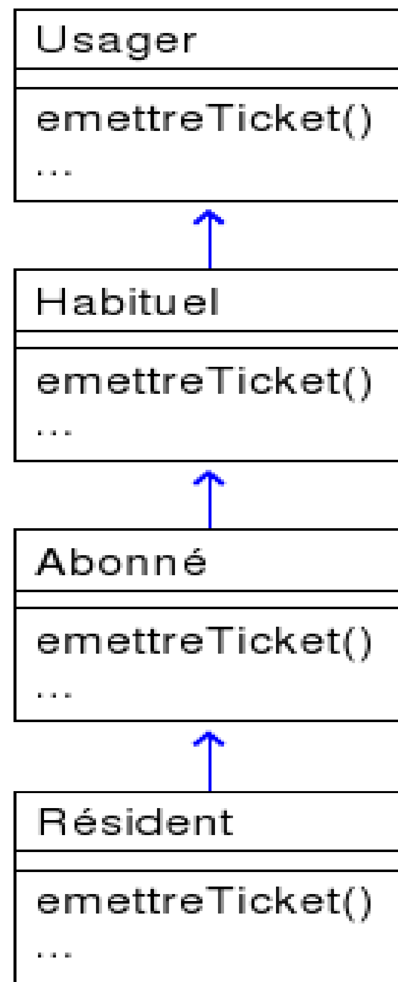
- des classes décrivant les utilisateurs du parking d'un immeuble

```
class Usager {
   tout vehicule se trouvant dans le parking
}
class Habituel extends Usager {
   usager frequent qui paye avec une carte de fidelite
}
class Abonne extends Habituel {
   les abonnes payent par ailleurs
}
class Resident extends Abonne {
   les residents ne payent pas
}
```

- la variable `unClient` représente un utilisateur précis :

```
Usager unClient ;
...
code donnant une valeur à la variable unClient
```

Polymorphisme et méthodes virtuelles



- l'expression rouge :

```
Usager unClient ;
...
unClient.emettreTicket() ;
```

  pose deux questions :

- *cette expression passe-t-elle la compilation ?*

  Oui, parce que la méthode `emettreTicket()` est membre de la classe `Usager`

- *peut-on dire quel sera son effet à l'exécution ?*

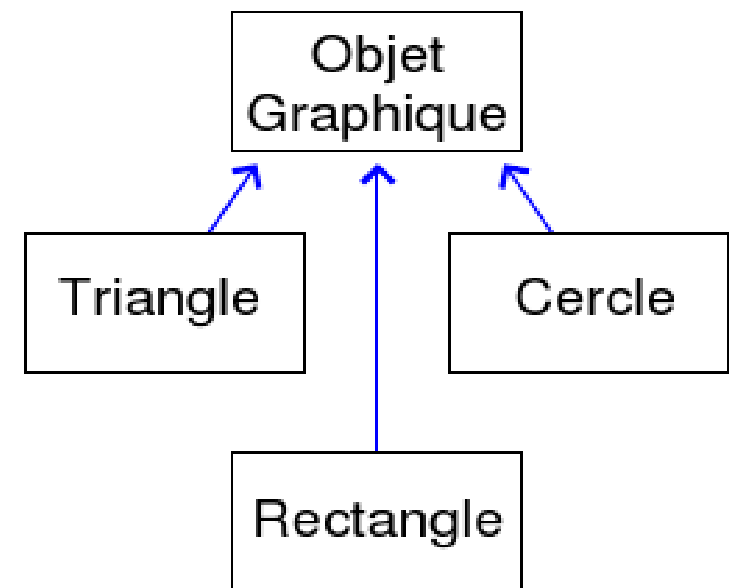
  Non, car la méthode `emettreTicket()` est redéfinie l'effet précis dépendra donc de la valeur *effective* de `unClient` au moment de l'appel.

- *A retenir* : la méthode appelée dans `unObjet.uneMethode(...)` est toujours la plus « basse » possible, c.-à-d. la plus proche du type effectif de unObjet

Polymorphisme et méthodes virtuelles

- L'exemple magique

```
class ObjetGraphique {
    public void seDessiner() {
        System.out.println("ERREUR") ;
    }
}
class Triangle extends ObjetGraphique {
    public void seDessiner() {
        opérations pour dessiner un triangle
    }
}
class Cercle extends ObjetGraphique {
    public void seDessiner() {
        opérations pour dessiner un cercle
    }
}
class Rectangle extends ObjetGraphique {
    public void seDessiner() {
        opérations pour dessiner un rectangle
    }
}
```



Polymorphisme et méthodes virtuelles

- Une image (complexe) est une collection d'objets graphiques
- Déclaration

```
ObjetGraphique[] image
   = new ObjetGraphique[n] ;
```

- Construction

```
image[0] = new Triangle(...) ;
image[1] = new Cercle(...) ;
image[2] = new Rectangle(...) ;
...
```

- Affichage

```
for (int i = 0 ; i < n ; i++)
   image[i].seDessiner() ;
```