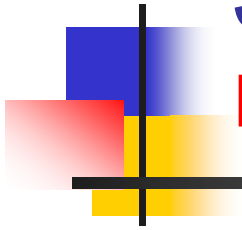
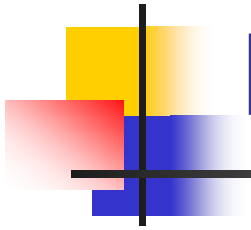


Java

Licence professionnelle CISII, 2009-2010

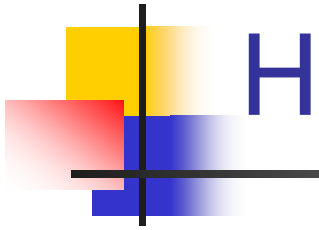
Cours 5 : l'héritage

Cours inspiré des cours de Richard Grin, Antoine Tabbone et Hazel Everett



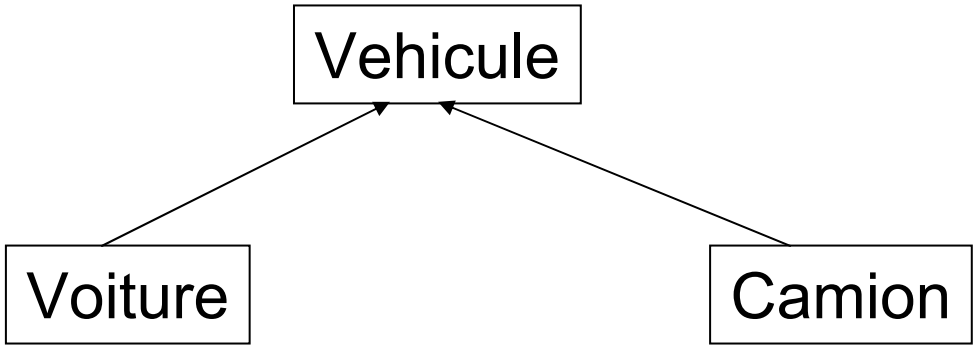
Héritage

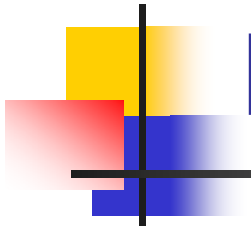
- Introduction
  - Pour raccourcir les temps d'écriture et de mise au point du code d'une application, il est intéressant de pouvoir réutiliser du code déjà écrit
  - Exemple
    - La classe Voiture représente toutes sortes de voitures possibles
    - On pourrait définir un camion comme une voiture très longue, très haute, etc.
    - Mais un camion a des spécificités par rapport aux voitures : remorque, cargaison, boîte noire, etc.
    - On pourrait créer une classe Camion qui ressemble à la classe Voiture
    - Mais on ne veut pas réécrire tout ce qu'elles ont en commun



Héritage

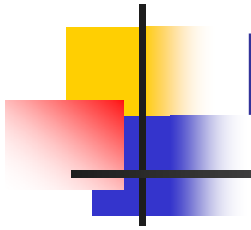
- Solution
  - La classe Vehicule contient tout ce qu'il y a de commun à Camion et Voiture
  - Camion ne contient que ce qu'il y a de spécifique aux camions





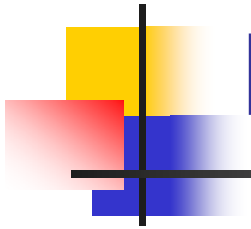
Héritage

- Objectif de l'héritage
  - Ne décrire qu'une seule fois le même traitement lorsqu'il s'applique à plusieurs classes
  - Évite de recopier (notamment les modifications)
  - Pour cela,
    - on crée une classe plus générique à laquelle s'applique le traitement
    - Toutes les classes plus spécifiques, héritant de cette classe, héritent de ce traitement, et peuvent l'exécuter
    - Le traitement n'est décrit qu'au niveau de la classe mère
    - Les classes filles contiennent d'autres traitements plus spécifiques



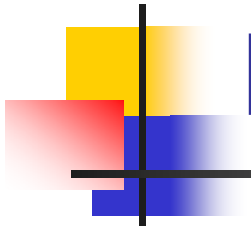
Héritage

- Usage de l'héritage
  - Une classe spécifique hérite des méthodes et des attributs de sa classe mère (sauf ceux qui sont privés)
  - On n'a pas besoin de les réécrire pour la classe fille
  - On peut cependant redéfinir une méthode de la classe mère dans la classe fille (de même signature)
  - Le constructeur d'un objet doit toujours commencer par appeler le constructeur de sa classe mère



Héritage

- Usage de l'héritage (suite)
  - Un objet de type Voiture peut utiliser toutes les méthodes de la classe Vehicule
  - Il doit disposer d'une valeur pour tous les attributs de la classe Vehicule
  - A tout moment, une méthode qui utilise un objet de type Vehicule peut manipuler un objet de type Voiture en guise de Vehicule
  - Cette dernière propriété est le polymorphisme



Héritage

- Exemple 1

```
class Vehicule {
      // Vehicule() {}
   }

class Voiture extends Vehicule {
   int nbPortes;
   double longueur;

    Voiture(double lg, int nbP)
     {
              longueur = lg;
              nbPortes = nbP;
     }
}
```

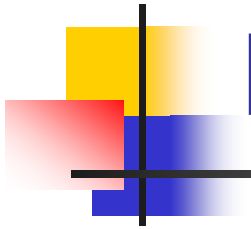
- Exemple 1 (suite)

```
class Garagiste {
    public boolean garer(Vehicule v)
     {
            v.demarrer();
            for (int pl=0;pl<nbPlaces;++pl)
                {
                    if (place[pl].estLibre())
                        {
                            v.amener(place[pl]);
                            v.arreter();
                            return true;
                        }
                }
            System.out.println("Aucune place libre");
            return false;
    }
}
```

On peut appliquer la méthode garer() à une Voiture, un Camion...

Un autre exemple

```
class Vehicule {
    String couleur ;
    int nbRoues ;
    boolean seDeplace ;
}
class VehiculeAMoteur extends Vehicule {
int puissance ;
}
class Autoroute {
    public static void main (String arg[]) {
      Vehicule v = new Vehicule () ;
      v.couleur = "rouge" ;
      v.nbRoues = 2 ;

      VehiculeAMoteur vm = new VehiculeAMoteur();
      vm.couleur = "bleu";
      vm.nbRoues = 4 ;
      vm.puissance = 7 ;
    }
}
```



Héritage

- Vocabulaire
  - La classe **A** s'appelle une classe mère, classe parente ou super-classe
  - La classe **B** qui hérite de la classe **A** s'appelle une classe fille ou sous-classe

Exemple de réutilisation : TsPointA.java

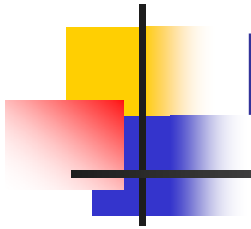
```
class Point
{ public void initialise (int x, int y) { this.x = x ; this.y = y ; }
  public void deplace (int dx, int dy)  { x += dx ; y += dy ; }
  public int getX() { return x ; }
  public int getY() { return y ; }
  private int x, y ;
}
class PointA extends Point
{ void affiche()
  { System.out.println ("Coordonnees : " + getX() + " " + getY()) ;
  }
}
public class TsPointA
{ public static void main (String args[])
  { Point p = new Point () ;
    p.initialise (2, 5) ;
    System.out.println ("Coordonnees : " + p.getX() + " " + p.getY() ) ;
    PointA pa = new PointA () ;
    pa.initialise (2, 5) ;  // on utilise la méthode initialise de Point
    pa.affiche() ;          // et la méthode affiche de PointA
  }
}
```

- Exemple de modification

```
public class Rectangle {
    private int x, y; // point en haut à gauche
    private int largeur, hauteur;

    // La classe contient des constructeurs,
    // des méthodes getX(), setX(int)
    // getHauteur(), getLargeur(),
    // setHauteur(int), setLargeur(int),
    // contient(Point), intersecte(Rectangle)
    // translateToi(Vecteur), toString(),...
    . . .
    public void dessineToi(Graphics g) {
      g.drawRect(x, y, largeur, hauteur);
    }
}
```

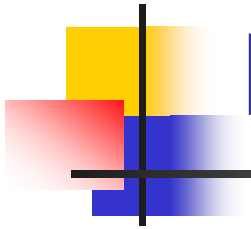
Exemple de classe fille

```
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.setCouleur(couleur);
      g.fillRect(getX(), getY(),
      getLargeur(), getHauteur());
    }
}
```



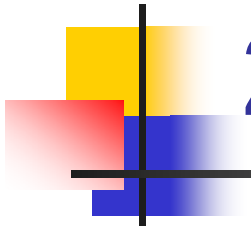
Héritage

- Code des classes filles
  - Quand on écrit la classe **RectangleColore**, on doit seulement
    - écrire le code (variables ou méthodes) lié aux nouvelles possibilités
      - on ajoute ainsi une variable **couleur** et les méthodes qui y sont liées
    - redéfinir certaines méthodes
      - on redéfinit la méthode **dessineToi()**



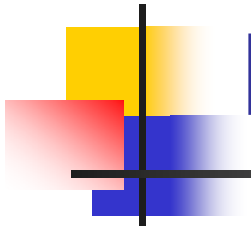
Héritage

- Redéfinition et surcharge
  - Ne pas confondre redéfinition et surcharge des méthodes :
    - On redéfinit une méthode quand une nouvelle méthode a la même signature qu'une méthode héritée de la classe mère
    - On surcharge une méthode quand une nouvelle méthode a le même nom, mais pas la même signature, qu'une autre méthode de la même classe
- Rappel
  - Signature d'une méthode (nom de la méthode + ensemble des types de ses paramètres)



2 façons de voir l'héritage

- Particularisation-généralisation :
  - Un rectangle coloré *est un* rectangle mais un rectangle particulier
  - La notion de figure géométrique est une généralisation de la notion de polygone
- 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 »

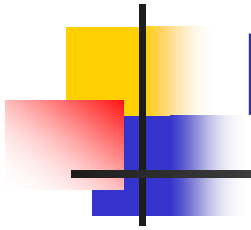


L’héritage en Java

- Fonctionnement
  - En Java, chaque classe a une et une seule classe mère (pas d’héritage multiple) dont elle hérite les variables et les méthodes
  - Le mot clef extends indique la classe mère :

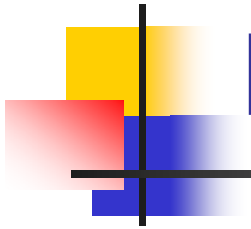
    ```
    class RectangleColore extends Rectangle
    ```

  - Par défaut (pas de extends dans la définition d’une classe), une classe hérite de la classe Object



Héritage

- Ce que peut faire une classe fille ?
  - La classe qui hérite peut
    - ajouter des variables, des méthodes et des constructeurs
    - redéfinir des méthodes (exactement les mêmes types de paramètres)
    - surcharger des méthodes (même nom mais pas même signature) (possible aussi à l'intérieur d'une classe)

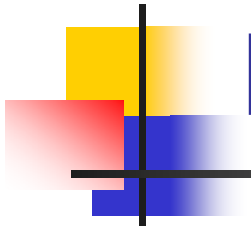


Héritage

- 1ère instruction d'un constructeur
  - La première instruction (interdit de placer cet appel ailleurs !) d'un constructeur peut être un appel
    - à un constructeur de la classe mère :

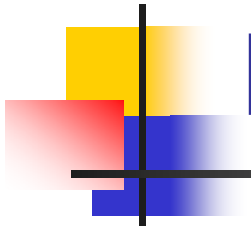
      super(...)
    - ou à un autre constructeur de la classe :

      this(...)



Héritage

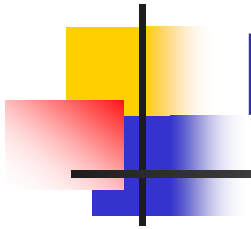
- 1ère instruction d'un constructeur
  - super() permet d'appeler le constructeur de la classe mère
  - C'est la première chose à faire dans la construction d'une sous-classe
  - Appeler le constructeur de la classe mère garantit que l'on peut initialiser les arguments de la classe mère
  - On passe les paramètres nécessaires
  - Si l'on n'indique pas super(), il y a un appel du constructeur par défaut de la classe mère



Héritage

- Constructeur de la classe mère

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



Héritage

- Constructeurs de la classe fille

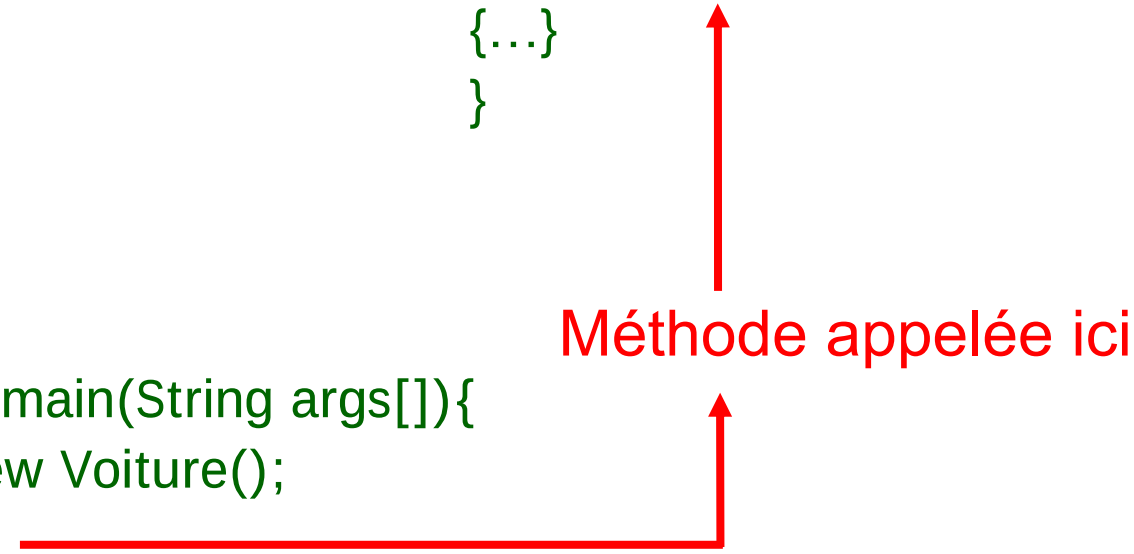
```
public class RectangleColore extends Rectangle {
private Color couleur;

    public RectangleColore(int x, int y,int largeur, int hauteur
    Color couleur) {
      super(x, y, largeur, hauteur); //appel explicite
      this.couleur = couleur;
    }
    public RectangleColore(int x, int y, int largeur, int hauteur) {
      this(x, y, largeur, hauteur, Color.black);
    }
    . . .
}
```

Appel de méthodes (1)

```
class Vehicule {
void demarrer()
{...}
}
```

```
class Voiture extends Vehicule{
void demarrer()
{…}
}
```

```
class Main {
public static void main(String args[]){
    Voiture v = new Voiture();
    v.demarrer();
    }
}
```



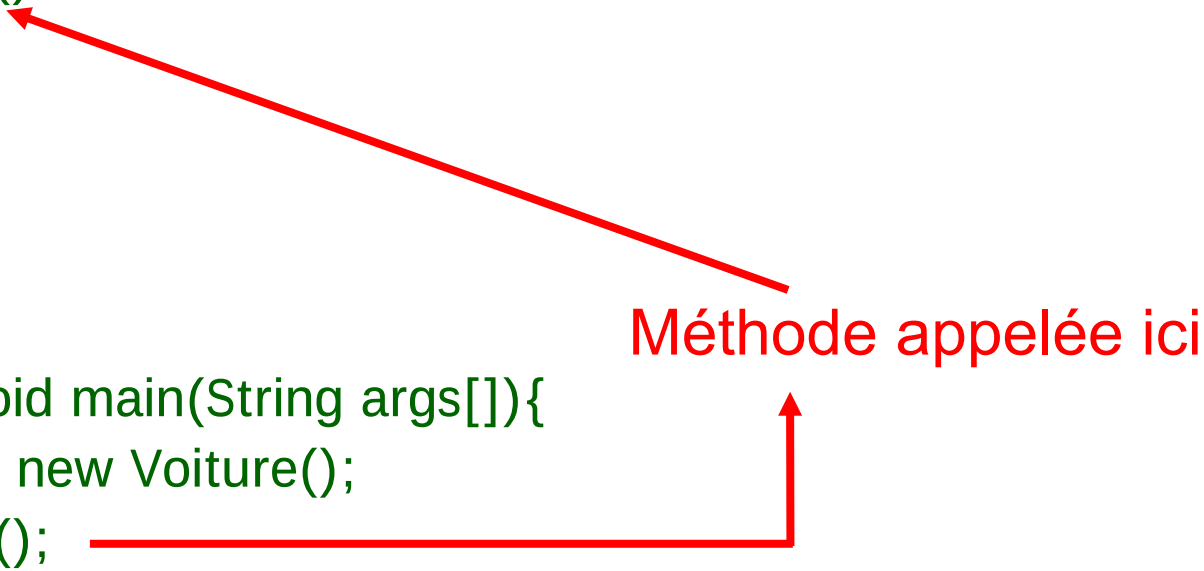
- Java commence par chercher la méthode dans la classe de l'objet créé

Appel de méthodes (2)

```
class Vehicule {
void demarrer()
{...}
}
```

```
class Voiture extends Vehicule{
}
```

```
class Main {
public static void main(String args[]){
    Voiture v = new Voiture();
    v.demarrer();
    }
}
```



- Si la méthode n'existe pas dans la classe de l'objet créé, il va chercher dans la classe mère

Appel de méthodes (3)

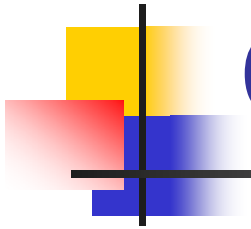
```
class Vehicule {
void demarrer()
{...}
}
```

```
class Voiture extends Vehicule{
void demarrer()
{…}
}
```

```
class Main {
public static void main(String args[]){
    Voiture v = new Voiture();
    v.super.demarrer();
    }
}
```

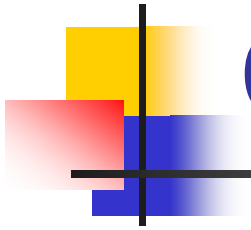
Méthode appelée ici

- Pour appeler directement la méthode de la classe mère, utiliser le mot clef super



Constructeur

- Appel implicite du constructeur de la classe mère
  - Si la première instruction d'un constructeur n'est ni **super(...)**, ni **this(...)**, le compilateur ajoute un appel implicite au constructeur sans paramètre de la classe mère (erreur s'il n'existe pas)
    - ➔ Un constructeur de la classe mère est toujours exécuté avant les autres instructions du constructeur



Constructeur

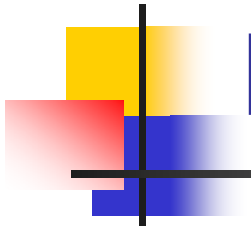
- Mais la première instruction d'un constructeur de la classe mère est l'appel à un constructeur de la classe « grand-mère », et ainsi de suite...

- Donc la toute première instruction qui est exécutée par un constructeur est le constructeur (sans paramètre) par défaut de la classe **Object** !

(D'ailleurs c'est le seul qui sait comment créer un nouvel objet en mémoire)

```
public class A {
  public A() {
    System.out.println("A");
  }
}

public class B extends A{
  public B() {
    System.out.println("B");
  }
}

public class C extends B {
  public C() {
    System.out.println("C");
  }

  public static void main(String args[]) {
    C c = new C();
  }
}
```

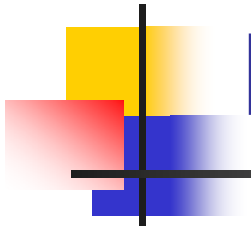
Question: Quel est le résultat?



Héritage

- Exemple d'appel implicite du constructeur de la classe mère

```
class Vehicule {
      // Vehicule() {}
    }

class Voiture extends Vehicule {
    int nbPortes;
    double longueur;

     Voiture(double lg, int nbP)
      {
            // super(); appel implicite de Vehicule(), sans paramètres
            longueur = lg;
            nbPortes = nbP;
      }
}
```



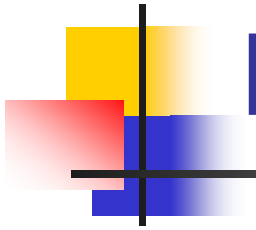
Héritage

Exemple sur les constructeurs

```
public class Cercle {
    // Constante
    public static final double PI = 3.14;
    // Variables
    private Point centre;
    private int rayon;
    // Constructeur
    public Cercle(Point c, int r) {
      centre = c;
      rayon = r;
    }
```

Ici pas de constructeur sans paramètre

Appel implicite du constructeur **Object()**



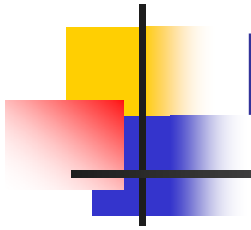
Héritage

```
// Méthodes
    public double surface() {
    return PI * rayon * rayon;
    }

    public Point getCentre() {
    return centre;
    }

    public static void main(String[] args) {
    Point p = new Point(1, 2);
    Cercle c = new Cercle(p, 5);
    System.out.println("Surface du cercle:"
    + c.surface());
    }
}
```

- Autre exemple d'appel implicite du constructeur de la classe mère

```
class Compteur {
    protected int v ;
    public Compteur (int v) {
      // exécution implicite de
    super(), c'est à dire le
    constructeur de Object qui ne
    fait rien
      this.v = v ;
    }
    public Compteur () {
      this (0) ;
      // appel de l'autre constructeur
    :
      Compteur (int v)
    }
    public void incr () {
      v++ ;
    } // ...
}
```

```
class CompteurMod extends
    Compteur {
  private int Mod ;
  public CompteurMod (int Mod) {
     // execution implicite de super(),
    c'est a dire Compteur ()
     this.Mod = Mod ;
  }
  public CompteurMod (int v, int
    Mod) {
     super (v % Mod) ; // appel
    explicite de Compteur (int v)
     this.Mod = Mod ;
  }
  public void incr () { v = (v + 1)%
    Mod ; } // redefinie ou overridden
  // ...
}
```



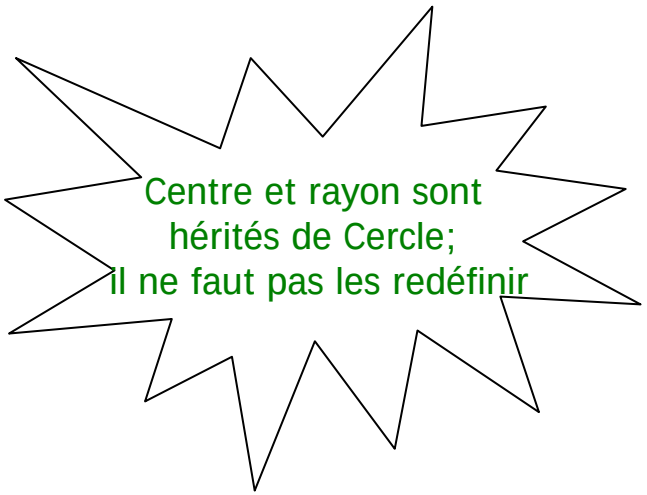
Héritage

```
public class CercleColore extends Cercle {
    private String couleur;

    public CercleColore(Point p, int r, String c) {
    super(p, r);
    couleur = c;
    }

    public void setCouleur(String c) {
    couleur = c;
    }
    public String getCouleur() {
    return couleur;
    }
}
```

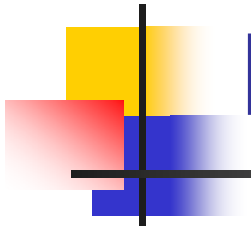
Que se passe-t-il si on enlève cette instruction ?

Héritage : erreur de débutant !

```
public class CercleColore extends Cercle {
    private Point centre;
    private int rayon;
    private String couleur;
    public CercleColore(Point p, int r, String c) {
      centre = c;
      rayon = r;
      couleur = c;
}
```

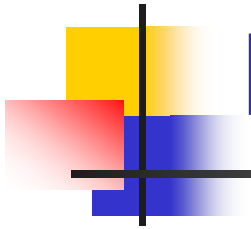


- Que se passe-t-il ici?



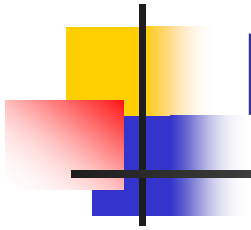
Exercice

- Énoncé
  - Écrire une classe Animal qui dispose d'un attribut entier nbPattes
  - Cette classe dispose des méthodes suivantes :
    - le constructeur, qui prend en argument un entier (le nombre de pattes)
    - String toString(), qui renvoie une chaîne de caractères contenant le nombre de pattes de l'animal
    - affiche() qui affiche le nombre de pattes de l'animal
  - Écrire une classe Autruche qui hérite de Animal
  - Écrire une classe Lapin qui hérite de Animal
  - Écrire une classe Main dans laquelle la méthode main() crée un lapin et une autruche



Héritage – problème d'accès

```
public class Animal {
    String nom; // pas private ; à suivre...
    public Animal() {
    }
    public Animal(String unNom) {
      nom = unNom;
    }
    public void setNom(String unNom) {
      nom = unNom;
    }
    public String toString() {
      return "Animal " + nom;
    }
}
```



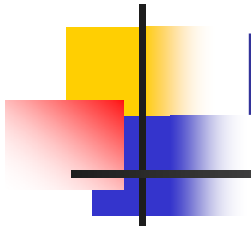
Héritage – problème d'accès

```
public class Poisson extends Animal {
    private int profondeurMax;
    public Poisson(String nom, int uneProfondeur) {
      this.nom = nom; // Et si nom est private ?
      profondeurMax = uneProfondeur;
    }
    public void setProfondeurMax(int uneProfondeur) {
      profondeurMax = uneProfondeur;
    }
    public String toString() {
      return "Poisson " + nom + " ; plonge jusqu'à "
      + profondeurMax + " mètres";
    }
}
```

Résolution par encapsulation

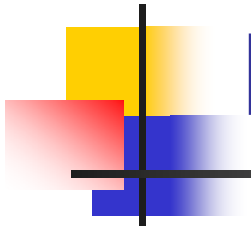
```
public class Poisson2 extends Animal {
    private int profondeurMax;
    public Poisson2(String unNom, int uneProfondeur) {
        super(unNom); // convient même si nom est private
        profondeurMax = uneProfondeur;
    }
    public void setProfondeurMax(int uneProfondeur) {
        profondeurMax = uneProfondeur;
    }
    public String toString() {
        return "Poisson " + getNom()
        + " plonge jusqu'à " + profondeurMax
        + " mètres";
    }
}
```

Accesseur obligatoire si nom est private



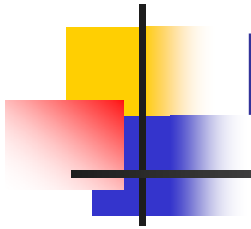
Héritage

- De quoi hérite une classe ?
  - Si une classe **B** hérite de **A** (**B extends A**), elle hérite automatiquement et implicitement de tous les membres de la classe **A** (mais pas des constructeurs)
  - Cependant la classe **B** peut ne pas avoir accès à certains membres dont elle a implicitement hérité de **A** (par exemple, les membres **private**)
  - Ces membres sont utilisés pour le bon fonctionnement de **B**, mais **B** ne peut pas les nommer ni les utiliser explicitement



Héritage

- Héritage et visibilité
  - Une classe fille hérite des attributs et méthodes public et protected de la classe mère
  - Elle n'hérite pas des attributs et méthodes private
  - La classe mère ne voit que ce qui est public dans la classe fille
  - Un objet d'une classe mère n'hérite pas des attributs et méthodes de ses classes filles

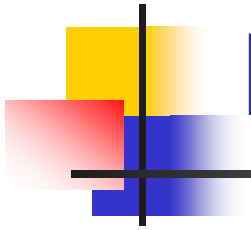


Héritage

- Héritage et visibilité : Exemple 1

```
public class Animal {
    protected String nom;
    . . .
}
public class Poisson extends Animal {
    private int profondeurMax;
    public Poisson(String unNom, int uneProfondeur) {
      nom = unNom; // utilisation de nom
      profondeurMax = uneProfondeur;
}
```

- On ne peut pas utiliser l'attribut nom dans la classe Poisson

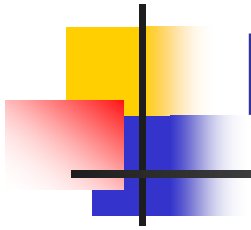


Héritage

- Héritage et visibilité : Exemple 2

```
class Vehicule {
      protected double longueur;
   }

class Voiture extends Vehicule {
   protected int nbPortes;
   ...
}
```

- On peut utiliser l'attribut longueur dans Vehicule et dans Voiture
- On ne peut pas utiliser l'attribut nbPortes dans la classe Vehicule



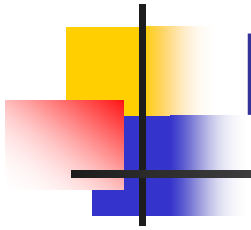
Héritage

- Polymorphisme
  - C'est un concept puissant de la POO qui complète l'héritage
  - Il explique comment une méthode peut se comporter différemment suivant l'objet sur lequel elle s'applique
  - Plus précisément, quand une même méthode est définie à la fois dans la classe mère et dans la classe fille, son exécution est réalisée en fonction de l'objet associé à l'appel et non plus suivant le nombre de paramètres, comme c'est le cas lors de la surcharge de méthodes

- Étudions l'exemple :

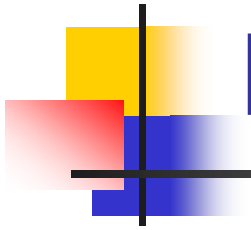
```
public class Forme {
 protected int x, y, couleur ;
 public Forme(int nx, int ny) {
    x = nx ;
    y = ny ;
    couleur = 0;
 }
 public void afficher() {
    System.out.println("Position en " + x + ", "
    + y);
    System.out.println("Couleur : " + couleur);
 }
 public void échangerAvec(Forme autre) {
    int tmp;
    tmp = x;
    x = autre.x;
    autre.x = tmp;
    tmp = y;
     y = autre.y;
    autre.y = tmp;
 }
 public void déplacer(int nx, int ny)   {
    x = nx;
    y = ny;
 }
} // Fin de la classe Forme
```

```
public class Cercle extends Forme {
    public final static int TailleEcran = 600 ;
    private int  r ;
    public Cercle(int xx, int yy){
            super(xx, yy);
           couleur = 10;
           r = rayonVérifié();
    }
    public void afficher()  {
           super.afficher();
           System.out.println("Rayon : " + r);
    }
    ...
}// Fin de la classe Cercle
```



Polymorphisme

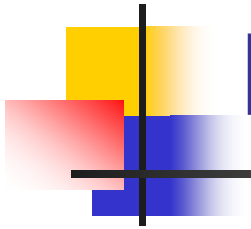
- Explications
  - La méthode afficher() est décrite dans la classe Forme et dans la classe Cercle
  - Cette double définition ne correspond pas à une véritable surcharge de fonction
  - En effet, les deux méthodes afficher() sont définies sans aucun paramètre
  - Le choix de la méthode ne peut donc s'effectuer sur la différence de paramètres
  - Il est effectuée par rapport à l'objet sur lequel la méthode est appliquée



Polymorphisme

- Observons l'exécution du programme suivant

```
public class FormerDesCercles {
 public static void main(String [] arg) {
   Cercle A  = new Cercle1(5, 5);
   A.afficher(); // appliquée à A
   Forme F = new Forme (10, 10);
   F.afficher(); //appliquée à F
 }
}
```



Polymorphisme

- Exemple : la situation
  - Considérons cette situation dans laquelle les classes Point et Poincol sont censées disposer chacune d'une méthode affiche()

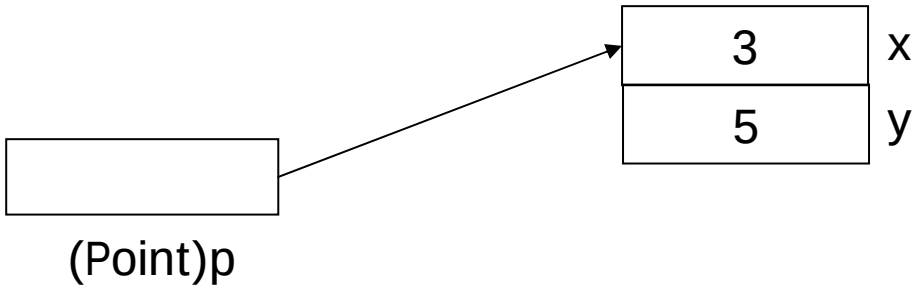
```
class Point {
   public Point(int x, int y){...}
   public void affiche() {...}
}

Class Pointcol extends Point {
   public Pointcol(int x, int y, byte couleur){...}
   public void affiche() {...}
}
```

Polymorphisme

- Exemple : référence
  - Avec ces instructions

    Point p;

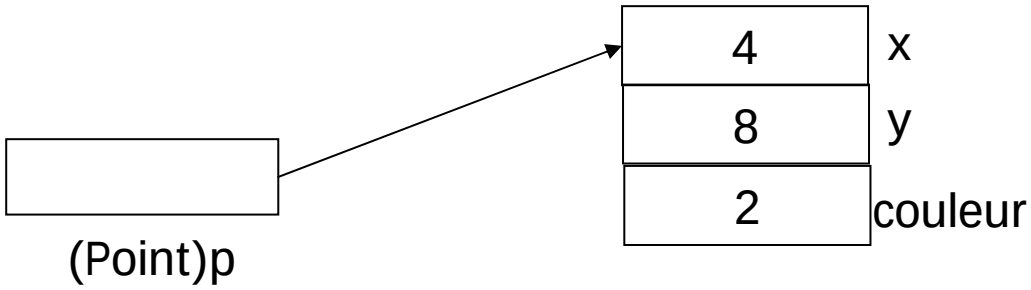
    p = new Point(3,5);
  - On aboutit tout naturellement à cette situation

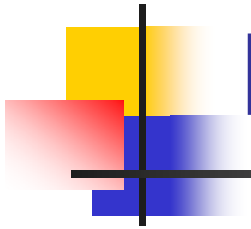


Polymorphisme

- Exemple : changement de référence
  - Mais il se trouve que Java autorise ce genre d'affectation (p est toujours de type Point !)

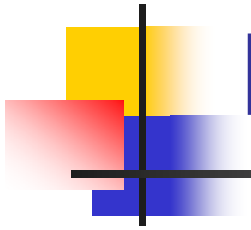
    p = new Pointcol(3,5, (byte)2)
  - La situation correspondante est :





Polymorphisme

- Exemple : constat
  - Java permet d'affecter à une variable objet non seulement la référence à un objet du type correspondant, mais aussi une référence à un objet d'un type dérivé
  - On est en présence d'une conversion implicite d'une référence à un type de classe T en une référence à un type ascendant de T



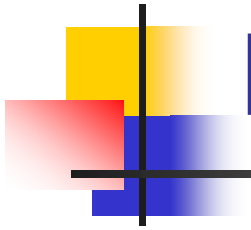
Polymorphisme

- Exemple : et pour affiche ?
  - Considérons maintenant ces instructions

```
Point p = new Point(3,5);
p.affiche(); //appelle la méthode affiche de la classe Point
p = new Pointcol(4,8,2);
p.affiche(); //appelle la méthode affiche de la classe Pointcol
```

  - Constat
    - Dans la dernière instruction, p est de type Point, alors que l'objet référencé par p est de type Pointcol
    - p.affiche appelle alors la méthode affiche de la classe Pointcol

  ➔ Java ne se fonde pas sur le type de la variable p mais bel et bien sur le type effectif de l'objet référencé au moment de l'appel



Polymorphisme

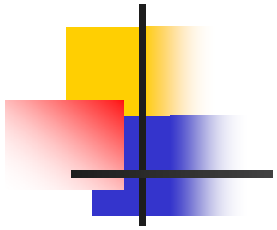
- Ligature dynamique
  - Ce choix d'une méthode au moment de l'exécution porte le nom de ligature dynamique
- En résumé
  - Le polymorphisme en Java se traduit par :
    - La compatibilité par affectation entre un type de classe et un type ascendant
    - La ligature dynamique des méthodes

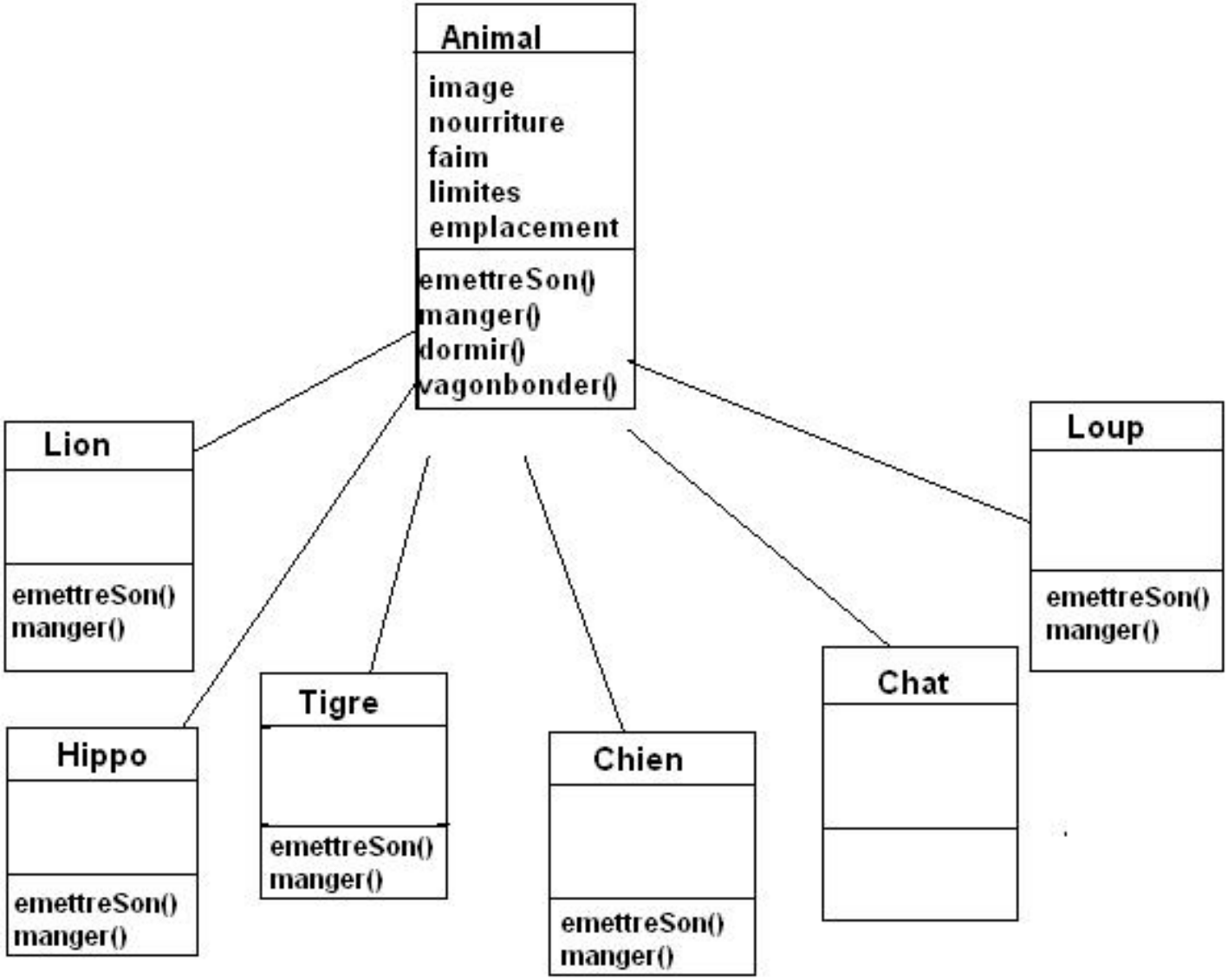
- Exemple complet intégrant les situations exposées ci-dessus

```
class Point2{
    private int x, y;
    public Point2 (int x, int y) {
      this.x = x ; this.y = y ; }
    public void deplace (int dx, int dy) {
      x += dx; y+=dy;}
    public void affiche(){
      System.out.println("Je suis en " + x +
    " " + y);}
 }

 class Pointcol extends Point2{
    private byte couleur;
    public Pointcol (int x, int y, byte
    couleur){
      super (x, y) ;
      this.couleur = couleur ;
    }
  public void affiche(){
        System.out.println("et ma couleur est :
        " +          couleur);
  }
}
```

```
public class Poly{
    public static void main (String
    args[]){
        Point2 p = new Point2 (3, 5) ;
        p.affiche(); //appelle affiche de
        Point
        Pointcol pc = new Pointcol(4,
        8, (byte)2) ;
        p = pc; //p de type Point,
        référence à un objet de type
        Pointcol
        p.affiche(); //appelle affiche de
        Pointcol
        p = new Point2 (5, 7); // p
        référence à nouveau un objet
        de type Point
        p.affiche(); //appelle affiche de
        Point
    }
}
```

Exemple :




Animal
image
nourriture
faim
limites
emplacement
emettreSon()
manger()
dormir()
vagonbonder()
Lion
emettreSon()
manger()
Loup
emettreSon()
manger()
Hippo
emettreSon()
manger()
Tigre
emettreSon()
manger()
Chien
emettreSon()
manger()
Chat

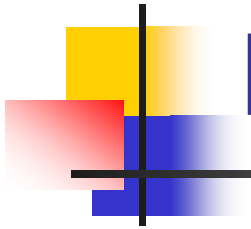
Polymorphisme

- Animal[] animaux=new Animal[5]
- Animaux[0]= new Chien();
- Animaux[1]= new Chat();
- Animaux[2]= new Loup();
- Animaux[3]= new Hippo();
- Animaux[4]= new Lion();

Déclare un tableau de type animal

On peut placer un objet de n'importe quelle sous-classe d'Animal dans le tableau

```
for (int i=0;i<animaux.length;i++) {
    Animaux[i].manger(); //Java saura faire la différence
    Animaux[i].vagonbonder(); //Java saura faire la différence
 }
```



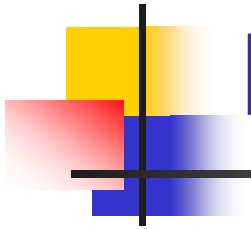
Polymorphisme

- Class Veto {

```
Public void soigner(Animal a) {
        a.emettreSon();
}
```

- Class ProprietaireDAnimaux {

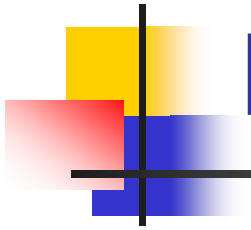
```
Public void start() {
        Veto v = new Veto();
        Chien c = new Chien();
        Hippo h = new Hippo();
        v.soigner(c);
        v.soigner(h);
}
```



Polymorphisme

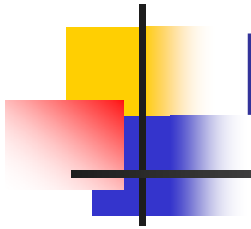
- Exemple 2

```
public class Figure {
    public void dessineToi() { }
}
public class Rectangle extends Figure {
    public void dessineToi() {
. . .
}
public class Cercle extends Figure {
    public void dessineToi() {
. . .
}
```



Polymorphisme

```
public class Dessin { // dessin composé de plusieurs figures
    private Figure[] figures;
    . . .
    public void afficheToi() {
    for (int i=0; i < nbFigures; i++)
      figures[i].dessineToi();
    }
    public static void main(String[] args) {
    Dessin dessin = new Dessin(30);
    . . . // création des points centre, p1, p2
    dessin.ajoute(new Cercle(centre, rayon));
    dessin.ajoute(new Rectangle(p1, p2));
    dessin.afficheToi();
...
```

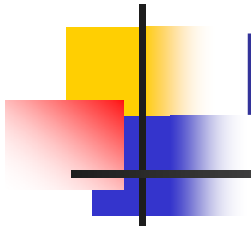


Polymorphisme

- Utilisation du polymorphisme
  - Bien utilisé, le polymorphisme évite les codes qui comportent de nombreux embranchements et tests

  - sans polymorphisme, la méthode **dessineToi()** aurait dû s'écrire :

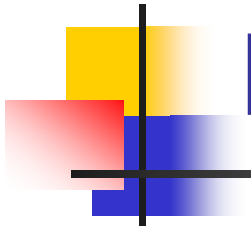
```
for (int i=0; i < figures.length; i++) {
        if (figures[i] instanceof Rectangle) {
                . . . / / dessin d'un rectangle
        }
        else if (figures[i] instanceof Cercle) {
                . . . / / dessin d'un cercle
        }
}
```



Polymorphisme

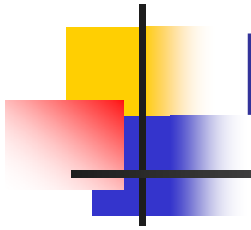
- Utilisation du polymorphisme
  - Le polymorphisme facilite l'extension des programmes :
    - on peut créer de nouvelles sous-classes sans toucher aux programmes déjà écrits
  - Par exemple, si on ajoute une classe **Losange**, le code de **afficheToi()** sera toujours valable
  - Sans polymorphisme, il aurait fallu modifier le code source de la classe **Dessin** pour ajouter un nouveau test :

```
if (figures[i] instanceof Losange) {
... // dessin d'un losange
```



Polymorphisme

- Extensibilité
  - Avec la programmation objet, une application est dite extensible si on peut étendre ses fonctionnalités **sans toucher au code source déjà écrit**
  - C’est possible en utilisant en particulier le polymorphisme comme on vient de le voir

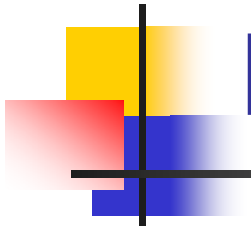


Polymorphisme

- *Transtypage ou Cast* : conversions de classes
  - Le « *cast* » est le fait de forcer le compilateur à considérer un objet comme étant d'un type qui n'est pas le type déclaré ou réel de l'objet

  - En Java, les seuls *casts* autorisés entre classes sont les *casts* entre classe mère et classes filles

  - On parle de *upcast* et de *downcast* *suivant le fait que le type est forcé de la classe fille vers la classe mère ou inversement*



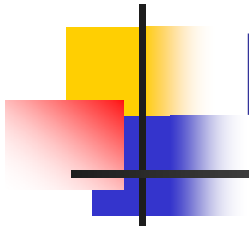
Polymorphisme

- Syntaxe
  - Pour caster un objet en classe C :

    ```
    (C) o;
    ```

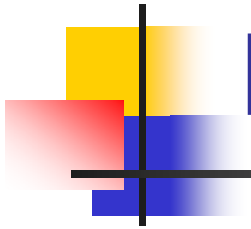
  - Exemple :

    ```
    Velo v = new Velo();
    Vehicule v2 = (Vehicule) v;
    ```



Polymorphisme

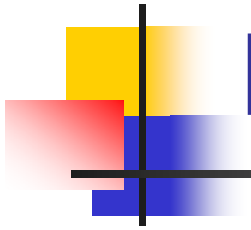
- *UpCast* : classe fille → classe mère
  - *Upcast* : un objet est considéré comme une instance d'une des classes ancêtres de sa classe réelle
  - Il est toujours possible de faire un *upcast* : à cause de la relation *est-un* de l'héritage, tout objet peut être considéré comme une instance d'une classe ancêtre
  - Le *upcast* est souvent implicite



Polymorphisme

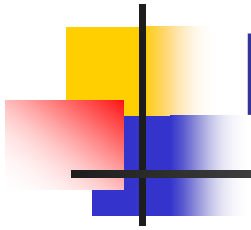
- Utilisation du *UpCast*
  - Il est souvent utilisé pour profiter ensuite du polymorphisme :

```
Figure[] figures = new Figure[10];
figures[0] = new Cercle(p1, 15);
. . .
figures[i].dessineToi();
```



Polymorphisme

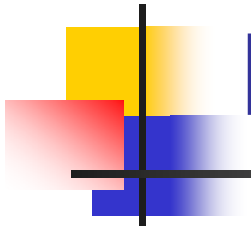
- *DownCast* : classe mère → classe fille
  - *Downcast* : un objet est considéré comme étant d'une classe fille de sa classe de déclaration
  - Toujours accepté par le compilateur, mais peut provoquer une erreur à l'exécution ; à l'exécution il sera vérifié que l'objet est bien de la classe fille
  - Un *downcast* doit toujours être explicite



Polymorphisme

- Utilisation du *DownCast*
  - Utilisé pour appeler une méthode de la classe fille qui n'existe pas dans une classe ancêtre

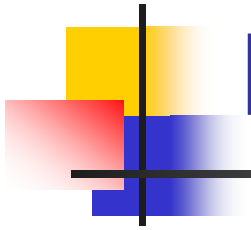
```
Figure f1 = new Cercle(p, 10);
. . .
Point p1 = ((Cercle)f1).getCentre();
```



Polymorphisme

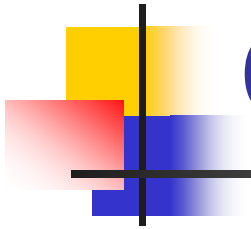
- *Downcast* pour récupérer les éléments d'une liste

```
// Ajoute des figures dans un ArrayList
// Un ArrayList contient un nombre quelconque
// d'instances de Object

ArrayList figures = new ArrayList();
figures.add(new Cercle(centre, rayon));
figures.add(new Rectangle(p1, p2));
. . .
// get() renvoie un Object. Cast nécessaire pour appeler
// dessineToi() qui n'est pas une méthode de Object

((Figure)figures.get(i)).dessineToi();
. . .
```



Polymorphisme

- Classe final (et autres au final)
  - Classe **final** :
    - ne peut avoir de classes filles (**String** est **final**)
  - Méthode **final** :
    - ne peut être redéfinie
  - Variable (locale ou d'état) **final** :
    - la valeur ne pourra être modifiée après son initialisation
  - Paramètre **final** (d'une méthode ou d'un **catch**) :
    - la valeur (éventuellement une référence) ne pourra être modifiée dans le code de la méthode



Cours5-TD5

- Exercices 1-2