

Programmation Orientée Objet

Héritage et polymorphisme

Frédéric Mallet

<http://deptinfo.unice.fr/~fmallet/>

Objectifs/Plan

☐ Notions essentielles

- Polymorphisme
- Classes abstraites

☐ Mécanismes

- Liaison dynamique
- Héritage
- Interfaces

« Regrouper DES objets de type différent par UNE classe qui rassemble les propriétés et opérations communes »

UNIFICATION

Les nombres complexes

Codage cartésien

```
class ComplexeCartesien {  
    double reelle;  
    double imaginaire;  
}
```

ComplexeCartesien

reelle : double
Imaginaire : double

Codage polaire

```
class ComplexePolaire {  
    double module;  
    double argument;  
}
```

ComplexePolaire

module : double
argument : double

Addition complexe:

$$(re1+i.im1) + (r2 + i.im2) = (re1+re2) + i.(im1+im2)$$

Addition complexe

□ Addition en Java

```
static ComplexeCartesien addition(ComplexeCartesien c1,  
                                   ComplexeCartesien c2) {  
    double reelle = c1.reelle + c2.reelle;  
    double imaginaire = c1.imaginaire + c2.imaginaire;  
    return new ComplexeCartesien(reelle, imaginaire);  
}
```

```
static ComplexeCartesien addition(ComplexePolaire c1,  
                                   ComplexePolaire c2) {  
    double reelle = c1.reelle() + c2.reelle();  
    double imaginaire = c1.imaginaire() + c2.imaginaire();  
    return new ComplexeCartesien(reelle, imaginaire);  
}
```

Addition complexe:

$$(re1+i.im1) + (r2 + i.im2) = (re1+re2) + i.(im1+im2)$$

Les interfaces

❑ Addition en Java

```
static ComplexeCartesien addition(IComplexe c1,
                                   IComplexe c2) {
    double reelle = c1.reelle() + c2.reelle();
    double imaginaire = c1.imaginaire() + c2.imaginaire();
    return new ComplexeCartesien(reelle, imaginaire);
}
```

« interface »
IComplexe

+ *reelle()* : double
+ *imaginaire()* : double

```
interface IComplexe {
    double reelle();
    double imaginaire();
}
```

Addition complexe:

$$(re1 + i.im1) + (r2 + i.im2) = (re1 + re2) + i.(im1 + im2)$$

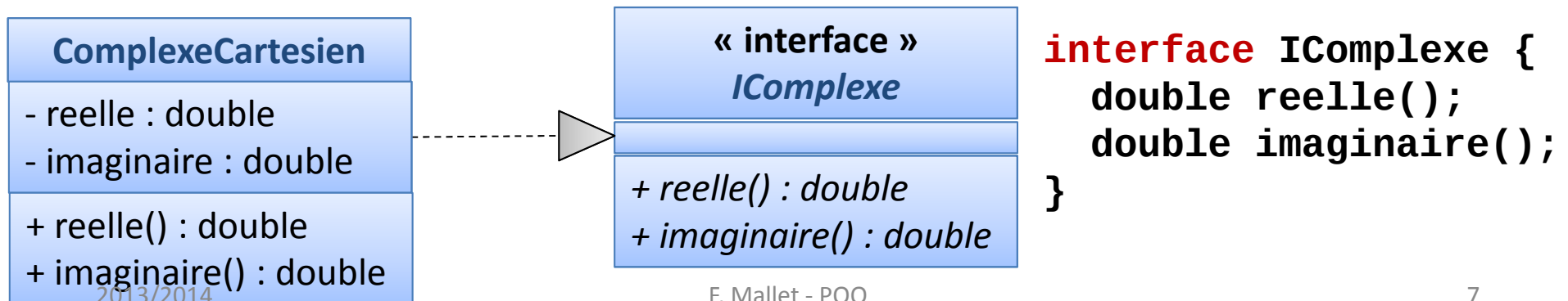
Planter une interface

❑ ComplexeCartesien **est-un** IComplexe

```
class ComplexeCartesien implements IComplexe {
    private double reelle;
    private double imaginaire;

    public double reelle() { return reelle; }

    public double imaginaire() { return imaginaire; }
}
```



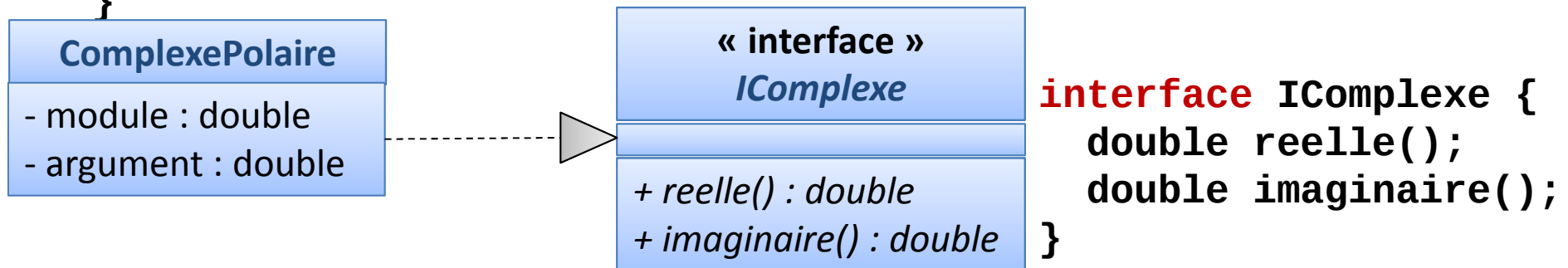
Planter une interface

❑ ComplexePolaire **est-un** IComplexe

```
class ComplexePolaire implements IComplexe {
    private double module, argument;

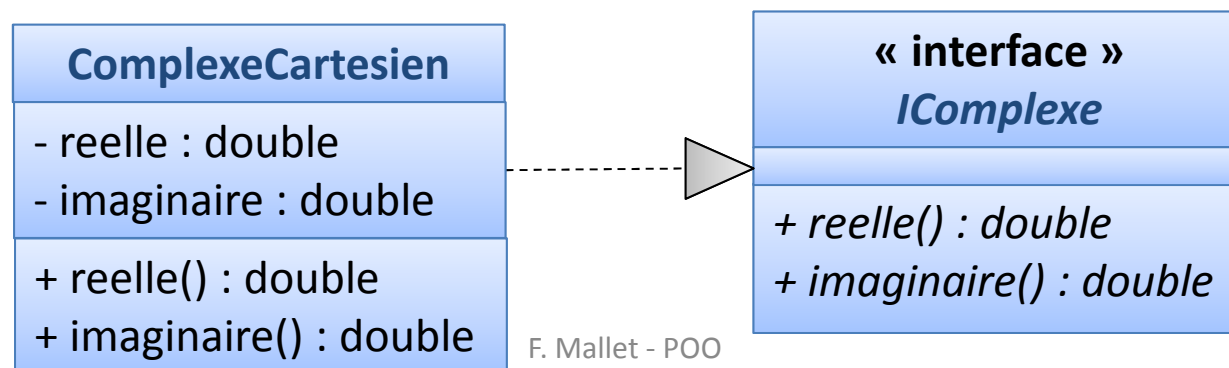
    public double reelle() {
        return module*module*Math.cos(argument);
    }

    public double imaginaire() {
        return module*module*Math.sin(argument);
    }
}
```



Polymorphisme

- ❑ Les références pointent des objets de **plusieurs** types
 - **IComplexe** c1 = new **ComplexeCartesien**(1, -1);
 - c1 est de **type statique** IComplexe
 - c1 référence un objet de **type dynamique** ComplexeCartesien
- ❑ Le type **statique**
 - Est choisi lors de la déclaration des références
- ❑ Le type **dynamique**
 - est choisi lors de l'affectation



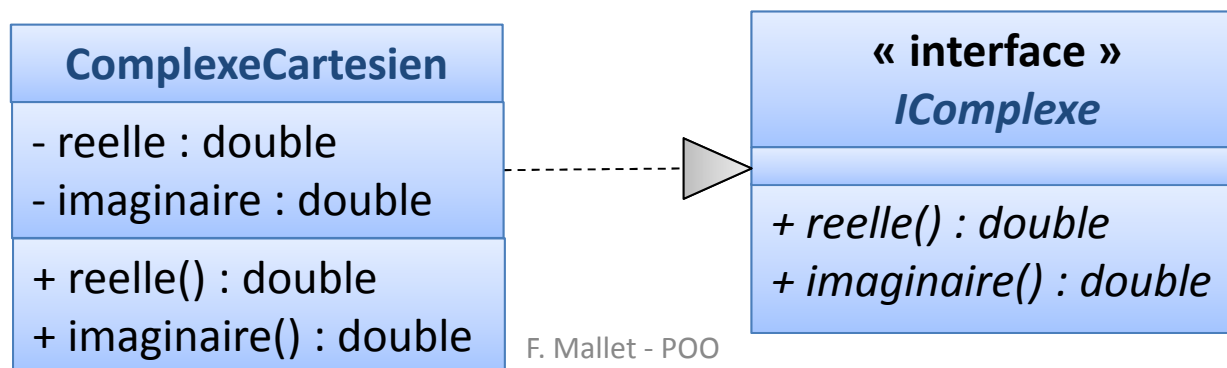
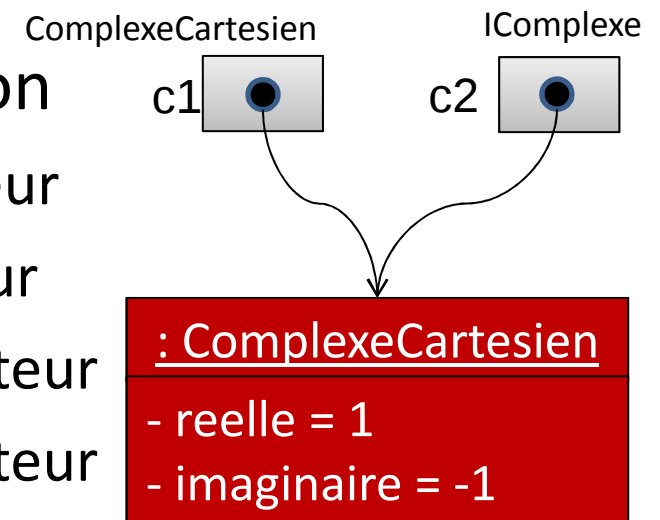
Polymorphisme

❑ Les références pointent des objets de **plusieurs** types

- **ComplexeCartesien** c1 = new **ComplexeCartesien**(1, -1);
- **Complexe** c2 = c1;

❑ Le type **statique** sert à la compilation

- **c1.reelle** est autorisé par le compilateur
- **c2.reelle** est interdit par le compilateur
- **c1.reelle()** est autorisé par le compilateur
- **c2.reelle()** est autorisé par le compilateur



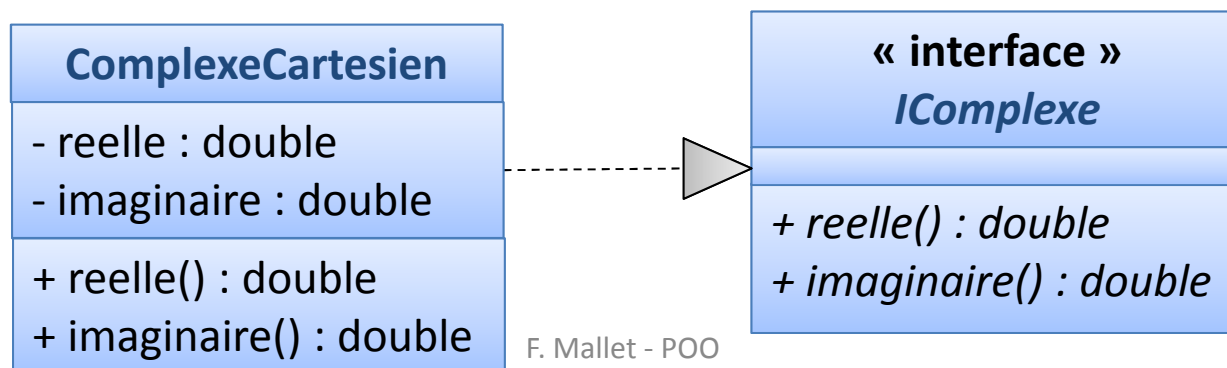
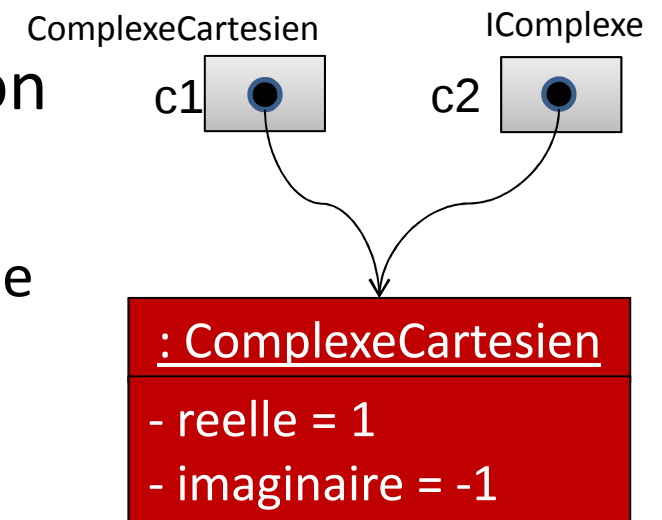
Liaison dynamique

❑ Les références pointent des objets de **plusieurs** types

- **ComplexeCartesien** c1 = new **ComplexeCartesien**(1, -1);
- **IComplexe** c2 = c1;

❑ Le type **dynamique** sert à l'exécution

- **c1.reelle()** et **c2.reelle()** provoque toutes les deux l'exécution de la méthode **reelle()** de la classe **ComplexeCartesien**



« Les classes peuvent représenter des types structurés, mais aussi des comportements »

ABSTRACTION

Méthodes d'instance et état

❑ Calculatrice.java

```
class Calculatrice {
    double accumulateur;
    void add(double v) {
        accumulateur += v;
    }
    void mul(double v) {
        accumulateur *= v;
    }
    void inv() {
        accumulateur = 1/accumulateur;
    }
    static final double PI = 3.1415;
}
```

Usage: `Calculatrice c = new Calculatrice();`
`c.add(12);`
`c.add(Calculatrice.PI);`



État : OUI

Ajout d'opérations ?

Les opérations binaires

❑ Qu'est-ce qu'une opération binaire réelle ?

```
interface IOperationBinaire {  
    double calcule(double c1, double c2);  
}
```

❑ L'addition est une opération binaire !

```
class Addition implements IOperationBinaire {  
    public double calcule(double c1, double c2) {  
        return c1 + c2;  
    }  
}
```

❑ Usage

```
IOperationBinaire op = new Addition();  
double v = op.calcule(12, 5);
```

Méthodes d'instance et état

❑ Calculatrice.java

```
class Calculatrice {  
    double accumulateur;  
    void applique(IOperationBinaire op, double v) {  
        accumulateur = op.calcule(accumulateur, v);  
    }  
    double inv() {  
        accumulateur = 1/accumulateur;  
    }  
    static final double PI = 3.1415;  
}
```

Usage: **Calculatrice c = new Calculatrice();**
c.applique(new Addition(), 12);

« Permet une classification hiérarchique des types »

HERITAGE

Les formes géométriques : **point**

□ Les points se caractérisent par

- leur position: x,y

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

// accesseurs de x et y

```
    public double getX() { return this.x; }  
    public double getY() { return this.y; }  
}
```

Point
- x : double - y : double
+ getX() : double + getY() : double

Les formes géométriques : **rectangle**

□ Un rectangle est caractérisé par

- Sa position d'ancrage: un point
- Sa largeur et sa hauteur

```
class Rectangle {  
    private Point ancrage;  
    private double largeur, hauteur;  
    Rectangle(Point p, double l, double h) {  
        this.ancrage = p;  
        this.largeur = l;  
        this.hauteur = h;  
    }  
}
```

Rectangle
- ancrage : Point
- largeur : double
- hauteur : double

Les formes géométriques : **cercle**

□ Un cercle est une forme géométrique

- Qui possède un rayon et un centre

```
class Cercle {  
    private Point centre;  
    private double rayon;  
    Cercle(Point centre, double rayon) {  
        this.centre = centre;  
        this.rayon = rayon;  
    }  
}
```

Cercle
- centre : Point - rayon : double

Héritage

- ❑ L'héritage permet de mutualiser des propriétés communes
 - Seulement si la sémantique est la même !
 - Pas seulement un problème de typage
- ❑ Propriétés communes entre les rectangles et les cercles ?
 - Des formes géométriques
 - Point ancrage
 - Rectangle : coin ?
 - Cercle : centre ?

Rectangle

- ancrage : Point
- largeur : double
- hauteur : double

Cercle

- centre : Point
- rayon : double

Les formes géométriques : unification

- ❑ L'unification de typage peut se faire par les interfaces
 - Permet de manipuler uniformément les rectangles et les cercles avec les mêmes références

```
interface IFormeGéométrique {  
}  
class Rectangle implements IFormeGéométrique {  
...  
}  
class Cercle implements IFormeGéométrique {  
...  
}  
IFormeGéométrique f1 = new Rectangle(...);  
f1 = new Cercle(...);
```

Polymorphisme !

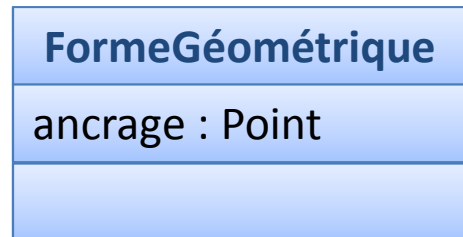
Héritage : mot clé **extends**

❑ Mutualiser des propriétés et des méthodes se fait par l'héritage

- Toute forme géométrique possède un point d'ancrage

```
class FormeGéométrique {  
    Point ancrage;  
}
```

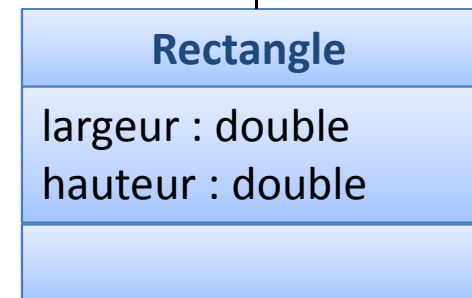
classe mère



- Un rectangle est une forme géométrique qui possède (en plus du point d'ancrage) une largeur et une hauteur.

```
class Rectangle extends FormeGéométrique {  
    double largeur, hauteur;  
}
```

classe fille



Classe **abstraite**

- ❑ Certaines classes représentent des concepts abstraits

```
abstract class AFormeGéométrique {  
    Point ancrage;  
}
```

- On peut avoir des références d'un type abstrait

```
AFormeGéométrique f1; // type statique
```

- Il est impossible d'instancier une classe abstraite !

```
f1 = new AFormeGéométrique();
```

- Il peut y avoir des objets de type AFormeGéométrique

```
f1 = new Rectangle(); // type dynamique
```

AFormeGéométrique

ancrage : Point



Rectangle

largeur : double
hauteur : double

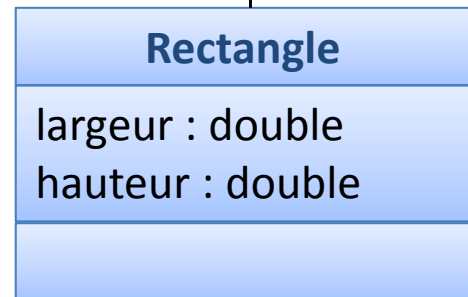
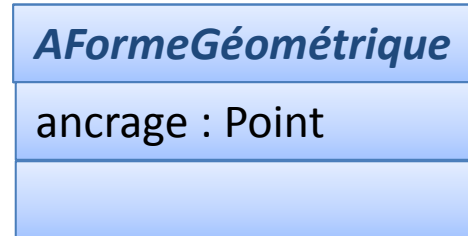
Héritage et constructeurs

- ❑ Chaque classe est responsable de l'initialisation de ses propriétés

```
abstract class AFormeGéométrique {  
    Point ancrage;  
    AFormeGéométrique(Point ancrage) {  
        this.ancrage = ancrage;  
    }  
}
```

- ❑ La classe fille doit appeler **UN** constructeur de sa classe mère

```
class Rectangle extends AFormeGéométrique {  
    double largeur, hauteur;  
    Rectangle(Point p, double largeur, double hauteur) {  
        super(p);  
        this.largeur = largeur;  
        this.hauteur = hauteur;  
    }  
}
```



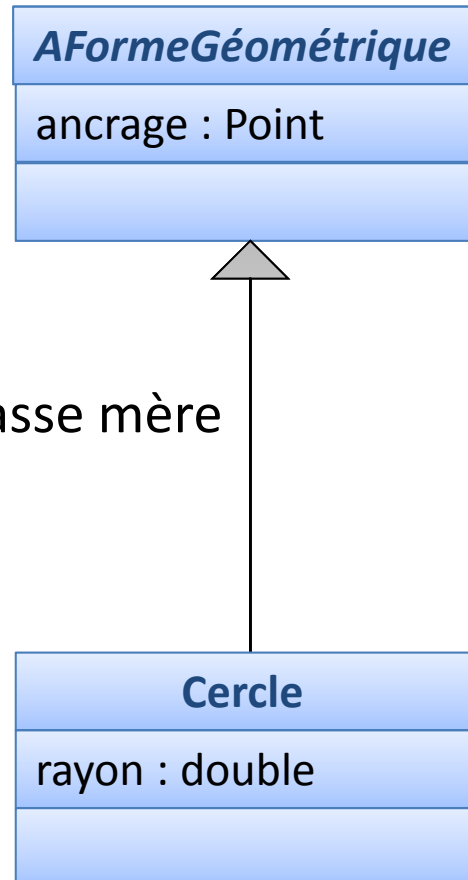
Héritage et constructeurs

- ❑ Chaque classe est responsable de l'initialisation de ses propriétés

```
abstract class AFormeGéométrique {  
    Point ancrage;  
    AFormeGéométrique(Point ancrage) {  
        this.ancrage = ancrage;  
    }  
}
```

- ❑ La classe fille doit appeler **UN** constructeur de sa classe mère

```
class Cercle extends AFormeGéométrique {  
    double rayon;  
    Cercle(Point p, double rayon) {  
        super(p);  
        this.rayon = rayon;  
    }  
}
```

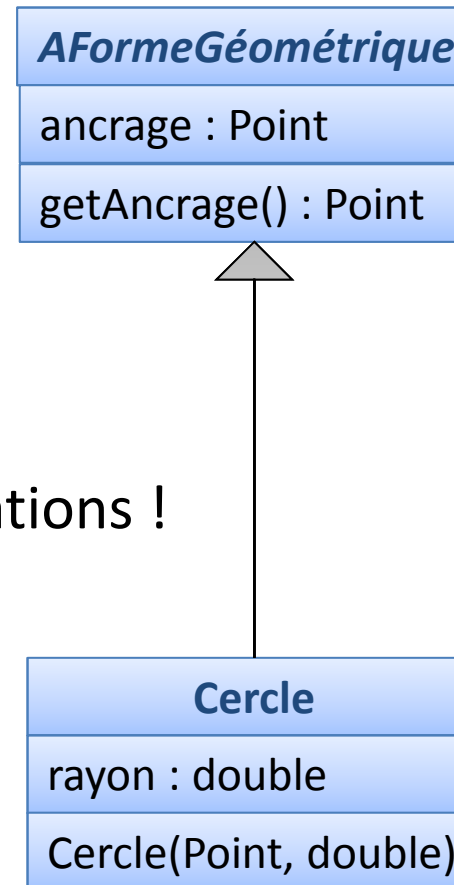


Héritage et membres

```
abstract class AFormeGéométrique {
    Point ancrage;
    AFormeGéométrique(Point ancrage) {
        this.ancrage = ancrage;
    }
    // accesseur
    Point getAncrage() { return ancrage; }
}
```

❑ Les classes héritent des propriétés et des opérations !

```
Cercle c = new Cercle(new Point(10,20), 40);
c.ancrage;
c.rayon;
c.getAncrage();
```



Héritage et polymorphisme

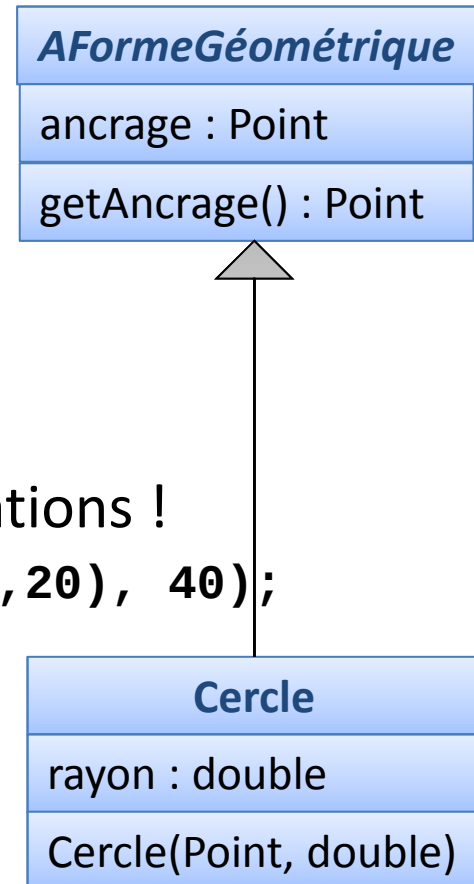
```

abstract class AFormeGéométrique {
    Point ancrage;
    AFormeGéométrique(Point ancrage) {
        this.ancrage = ancrage;
    }
    // accesseur
    Point getAncrage() { return ancrage; }
}
  
```

❑ Les classes héritent des propriétés et des opérations !

```

AFormeGéométrique f = new Cercle(new Point(10,20), 40);
f.ancrage;
f.rayon;
f.getAncrage();
  
```



Héritage et modificateurs

Formes géométriques abstraites

```
abstract class AFormeGéométrique {  
    Point ancrage;  
    AFormeGéométrique(Point ancrage) {  
        this.ancrage = ancrage;  
    }  
    // accesseur  
    Point getAncrage() {  
        return ancrage;  
    }  
}
```

Cercle

```
class Cercle  
    extends AFormeGéométrique {  
    double rayon;  
    Cercle(Point p, double r) {  
        super(p);  
        this.rayon = r;  
    }  
    Point getCentre() {  
        return super.ancrage;  
    }  
}
```

❑ Le mot clé **super** permet d'accéder à la classe mère

Héritage et modificateurs

Formes géométriques abstraites

```
abstract class AFormeGéométrique {  
    private Point ancrage;  
    AFormeGéométrique(Point ancrage) {  
        this.ancrage = ancrage;  
    }  
    // accesseur  
    Point getAncrage() {  
        return ancrage;  
    }  
}
```

Cercle

```
class Cercle  
    extends AFormeGéométrique {  
    private double rayon;  
    Cercle(Point p, double r) {  
        super(p);  
        this.rayon = r;  
    }  
    Point getCentre() {  
        return super.ancrage;  
    }  
}
```

- ❑ Les membres privés ne sont accessibles que dans le bloc de déclaration de classe !

Héritage et modificateurs

Formes géométriques abstraites

```
abstract class AFormeGéométrique {  
    protected Point ancrage;  
    AFormeGéométrique(Point ancrage) {  
        this.ancrage = ancrage;  
    }  
    // accesseur  
    Point getAncrage() {  
        return ancrage;  
    }  
}
```

Cercle

```
class Cercle  
    extends AFormeGéométrique {  
    private double rayon;  
    Cercle(Point p, double r) {  
        super(p);  
        this.rayon = r;  
    }  
    Point getCentre() {  
        return super.ancrage;  
    }  
}
```

- ☐ Les membres **protégés** sont accessibles à partir
 - ☐ Des blocs de déclaration des classes filles (transitivement)
 - ☐ Des classes appartenant au même paquetage

Héritage et modificateurs

Formes géométriques abstraites

```
abstract class AFormeGéométrique {  
    private Point ancrage;  
    AFormeGéométrique(Point ancrage) {  
        this.ancrage = ancrage;  
    }  
    // accesseur  
    Point getAncrage() {  
        return ancrage;  
    }  
}
```

Cercle

```
class Cercle  
    extends AFormeGéométrique {  
    private double rayon;  
    Cercle(Point p, double r) {  
        super(p);  
        this.rayon = r;  
    }  
    Point getCentre() {  
        return super.getAncrage();  
    }  
}
```

- ❑ Les membres **package** sont accessibles à partir
 - ❑ Des classes appartenant au même paquetage

COMPLÉMENTS

La classe `java.lang.Object`

- ❑ Toutes les classes héritent de `java.lang.Object`

```
abstract class AFormeGéométrique {  
    private Point ancrage;  
    AFormeGéométrique(Point ancrage) {  
        this.ancrage = ancrage;  
    }  
    // accesseur  
    Point getAncrage() {  
        return ancrage;  
    }  
}
```

AFormeGéométrique

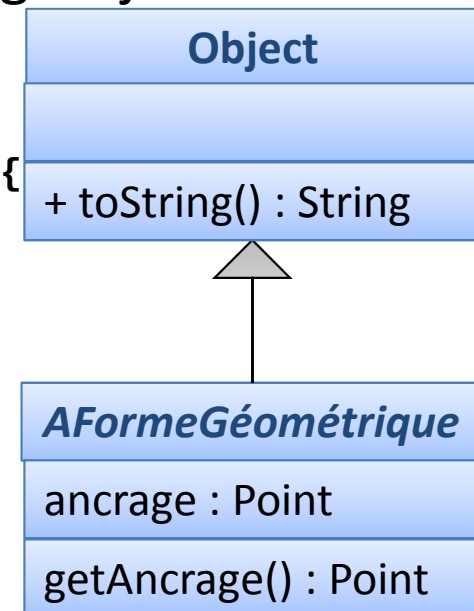
ancrage : Point

getAncrage() : Point

La classe `java.lang.Object`

- ❑ Toutes les classes héritent de `java.lang.Object`
 - Pas d'héritage équivaut à hériter de `java.lang.Object`

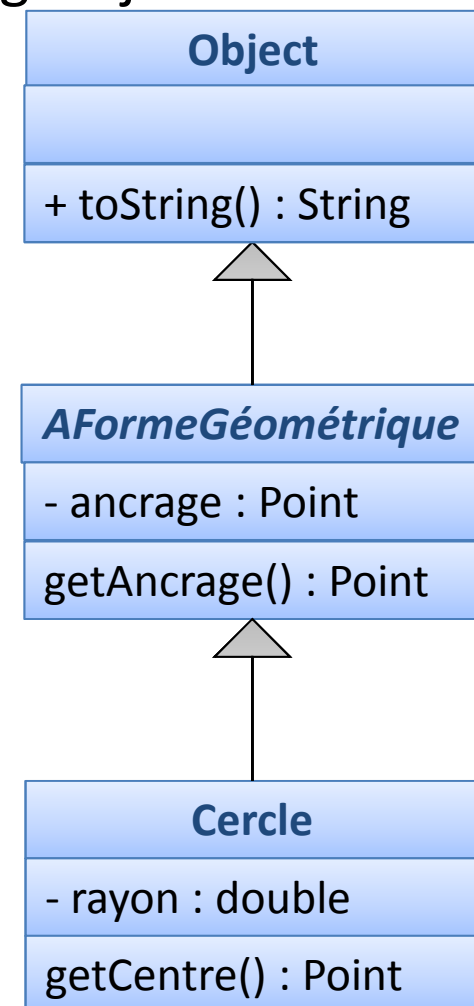
```
abstract class AFormeGéométrique extends java.lang.Object {  
    private Point ancrage;  
    AFormeGéométrique(Point ancrage) {  
        super();  
        this.ancrage = ancrage;  
    }  
    // accesseur  
    Point getAncrage() {  
        return ancrage;  
    }  
}
```



La classe `java.lang.Object`

- ❑ Toutes les classes héritent de `java.lang.Object`
 - Pas d'héritage équivaut à hériter de `java.lang.Object`
 - L'héritage est transitif

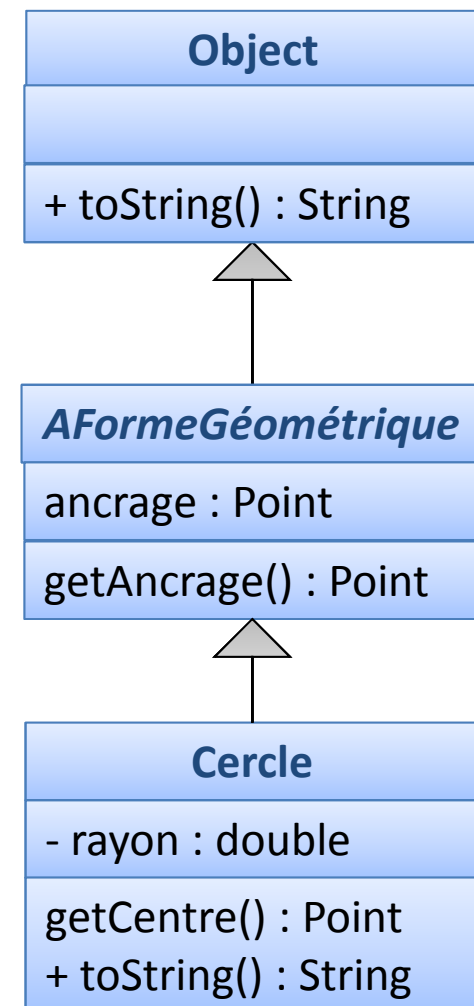
```
class Cercle extends AFormeGéométrique {  
    private double rayon;  
    Cercle(Point centre, double rayon) {  
        super(centre);  
        this.rayon = rayon;  
    }  
    Point getCentre() {  
        return super.getAncrage();  
    }  
}
```



Héritage et **redéfinition**

- ❑ La classe fille peut redéfinir les méthodes de sa mère
 - Il faut préserver la signature des méthodes redéfinies
 - Ce n'est pas de la surcharge !

```
class Cercle extends AFormeGéométrique {  
    private double rayon;  
    Cercle(Point centre, double rayon) {  
        super(centre);  
        this.rayon = rayon;  
    }  
    Point getCentre() {  
        return super.getAncrage();  
    }  
    // redéfinition de Object.toString()  
    public String toString() {  
        return "Cercle:" + rayon;  
    }  
}
```

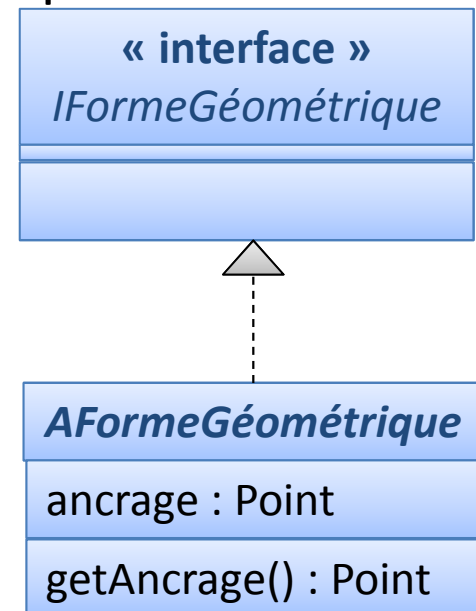


Compléments sur les interfaces

- ❑ L'implémentation d'interface est transitive
 - Les classes abstraites ne sont pas tenues d'implanter toutes les méthodes de leurs interfaces

```

abstract class AFormeGéométrique
                                implements IFormeGéométrique {
    private Point ancrage;
    AFormeGéométrique(Point ancrage) {
        super();
        this.ancrage = ancrage;
    }
    // accesseur
    Point getAncrage() {
        return ancrage;
    }
}
  
```



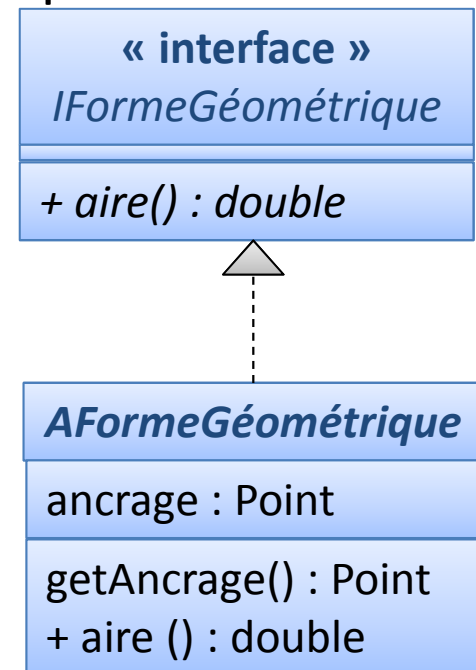
Compléments sur les interfaces

- ❑ L'implémentation d'interface est transitive
 - Les classes abstraites ne sont pas tenues d'implanter toutes les méthodes de leurs interfaces

```

abstract class AFormeGéométrique
    implements IFormeGéométrique {

    private Point ancrage;
    AFormeGéométrique(Point ancrage) {
        super();
        this.ancrage = ancrage;
    }
    // accesseur
    Point getAncrage() {
        return ancrage;
    }
    public double aire() { ? }
}
  
```



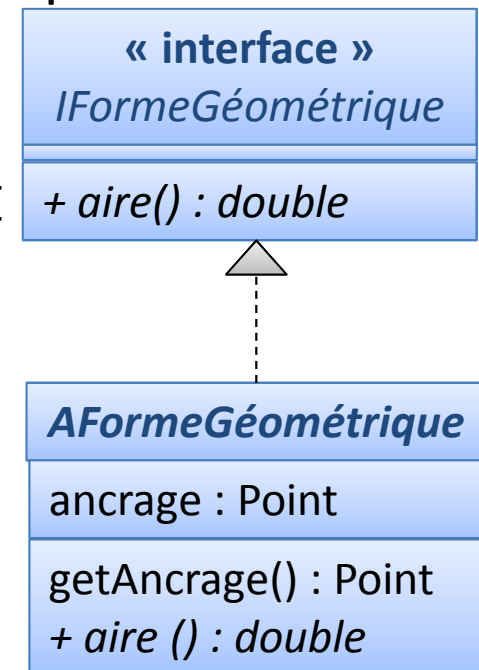
Compléments sur les interfaces

- ❑ L'implémentation d'interface est transitive
 - Les classes abstraites ne sont pas tenues d'implanter toutes les méthodes de leurs interfaces

```

abstract class AFormeGéométrique
    implements IFormeGéométrique {
    private Point ancrage;
    AFormeGéométrique(Point ancrage) {
        super();
        this.ancrage = ancrage;
    }
    // accesseur
    Point getAncrage() {
        return ancrage;
    }
    abstract public double aire();
}

```



Les formes géométriques

□ Une forme géométrique est un objet

- Pour lequel on peut calculer son aire

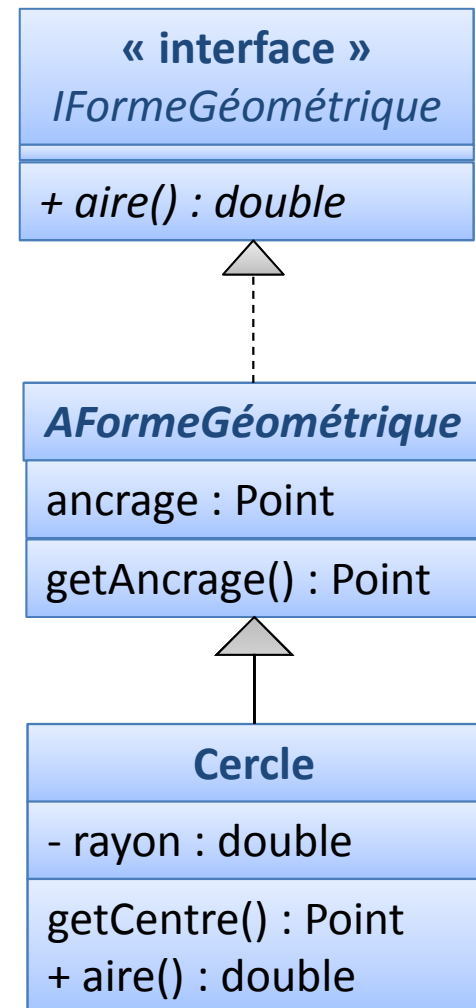
```
interface IFormeGéométrique {  
    double aire();  
}
```

« interface » <i>IFormeGéométrique</i>
+ <i>aire()</i> : double

Compléments sur les interfaces

- ❑ L'implémentation d'interface est transitive
 - Les classes concrètes DOIVENT implanter toutes les méthodes déclarées

```
class Cercle extends AFormeGéométrique {
    private double rayon;
    Cercle(Point centre, double rayon) {
        super(centre);
        this.rayon = rayon;
    }
    Point getCentre() {
        return super.getAncrage();
    }
    public double aire() {
        return Math.PI*rayon*rayon;
    }
}
```



Pour résumé

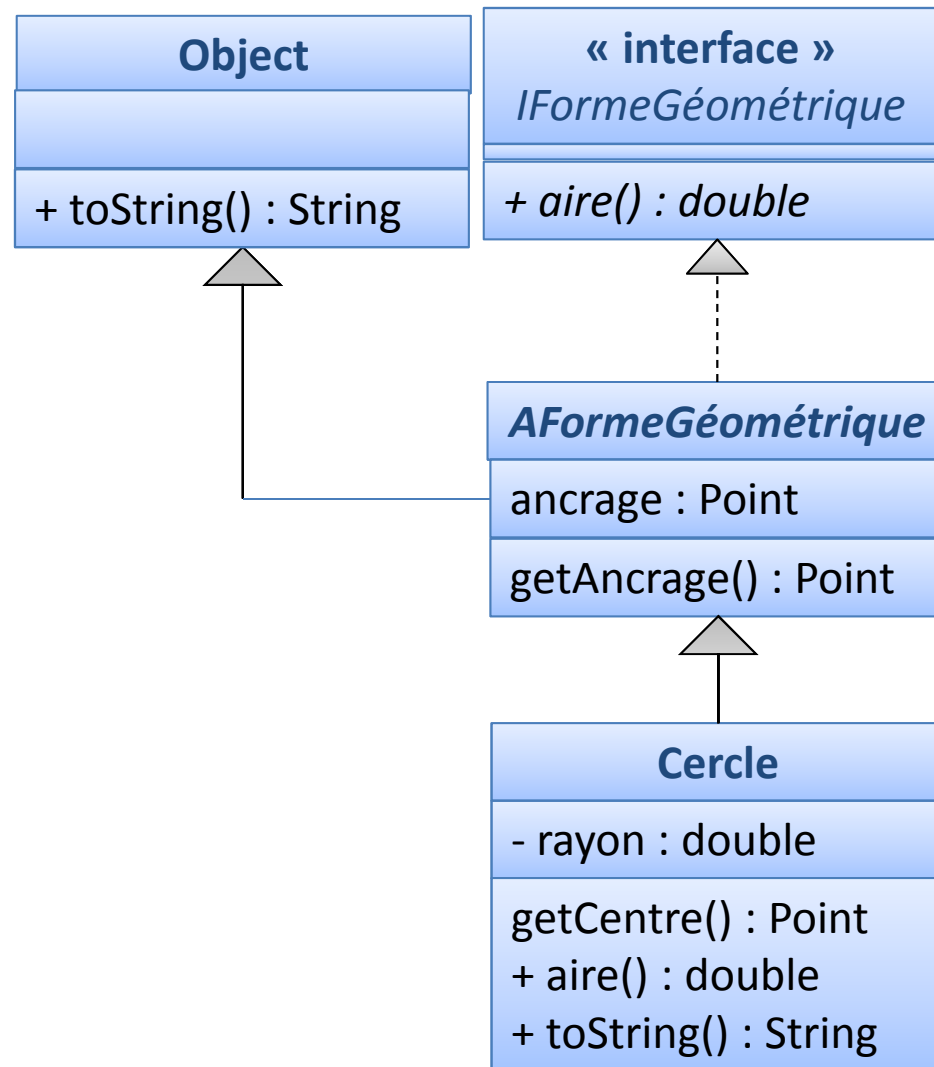
```
Point p = new Point(10, 20);
Cercle c = new Cercle(p, 40);
```

❑ Polymorphisme

```
Object o1 = p;
Object o2 = c;
IFormeGéométrique f1 = c;
IFormeGéométrique f2 = p;
AFormeGéométrique f3 = c;
```

❑ Type statique et compilation

```
c.toString(); 😊
p.toString(); 😊
o2 rayon;
f1 ancrage;
f3.ancrage; 😊
f3 getCentre();
c.ancrage; 😊
c. rayon;
c.getCentre(); 😊
```



Liaison dynamique

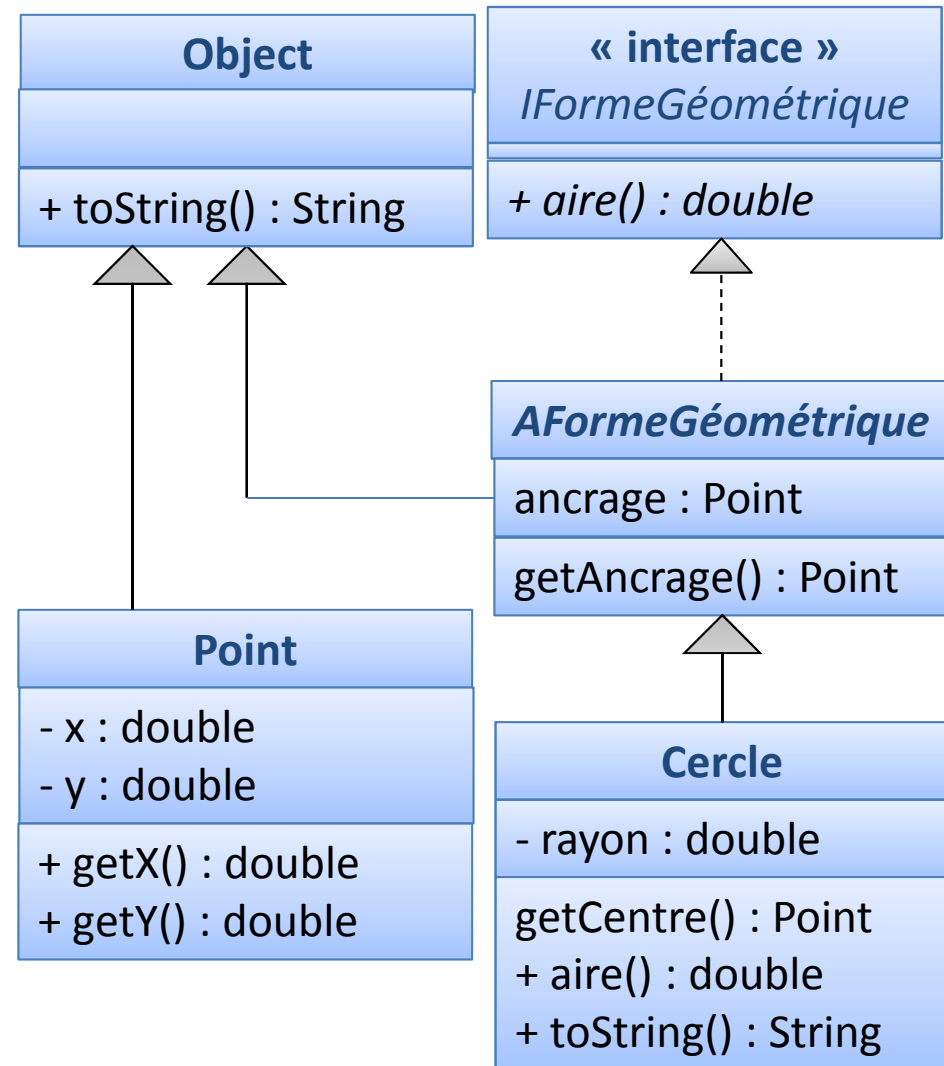
```
Point p = new Point(10, 20);
Cercle c = new Cercle(p, 40);
```

❑ Polymorphisme

```
Object o1 = p;
Object o2 = c;
IFormeGéométrique f1 = c;
AFormeGéométrique f3 = c;
```

❑ Type dynamique

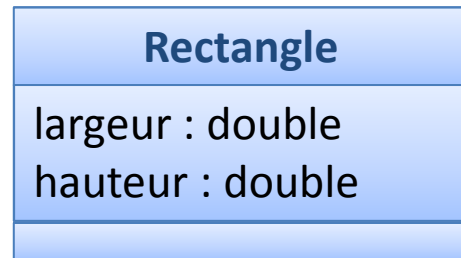
```
c.toString();
p.toString();
f1.aire();
c.getCentre();
c.getAncrage();
```



Préserver la sémantique de l'héritage

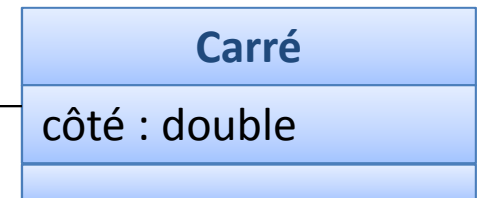
❑ Rectangle

```
class Rectangle {  
    double largeur;  
    double hauteur;  
}
```



❑ Carré

```
class Carré {  
    double côté;  
}
```



❑ Peut-on utiliser l'héritage?

- Si oui, comment ?

Préserver la sémantique de l'héritage

❑ Rectangle

```
class Rectangle {
    private double largeur;
    private double hauteur;
    Rectangle(double l, double h){
        this.largeur = l;
        this.hauteur = h;
    }
    double aire() {
        return largeur * hauteur;
    }
}
```

❑ Un carré **est un** rectangle

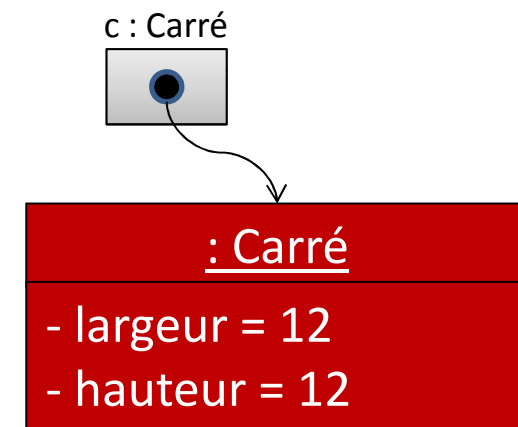
- Qui a les deux côtés de même longueur

❑ Carré

```
class Carré extends Rectangle {
    Carré(double cote) {
        super(cote, cote);
    }
}
```

❑ Usage

```
Carré c = new Carré(12);
c.aire();
```



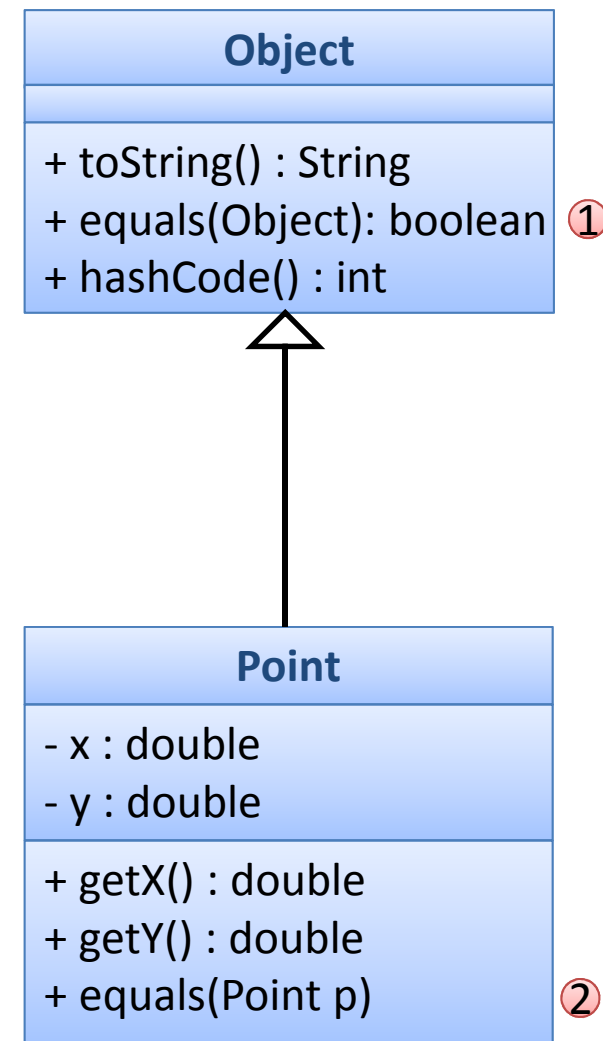
Surcharge et liaison dynamique

- ❑ Ne pas confondre surcharge et redéfinition

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

- ❑ Utilisation

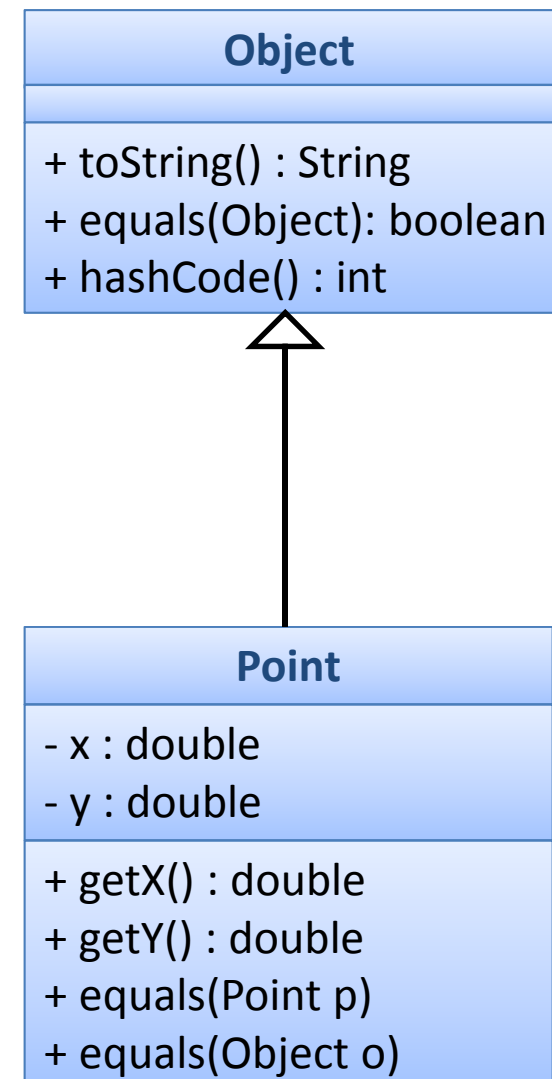
```
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);
Object o2 = p2;
p1.equals(o2);
p1.equals(p2);
o2.equals(p1);
p2.equals(p1);
```



Surcharge et liaison dynamique

❑ Ne pas confondre surcharge et redéfinition

```
class Point {  
    int x, y;  
    boolean equals(Point p) {  
        return p.x==this.x && p.y == this.y;  
    }  
  
    public boolean equals(Object o) {  
        if (o instanceof Point)  
            return equals((Point)o);  
        return false;  
    }  
  
    public int hashCode() {  
        return x<<16+y;  
    }  
}
```



Surcharge et liaison dynamique

❑ Ne pas confondre surcharge et redéfinition

❑ Utilisation

```
Point p1 = new Point(1,2);  
Point p2 = new Point(1,2);  
Object o1 = p1;  
Object o2 = p2;  
p1.equals(o2);  
p1.equals(p2);  
o1.equals(o2);  
o1.equals(p2);  
o2.equals(o1);  
o2.equals(p1);  
p2.equals(o1);  
p2.equals(p1);
```

