

# Les interfaces avec Java

Programmation objet - IV

L2 informatique

F. Bertrand

# Au programme d'aujourd'hui...

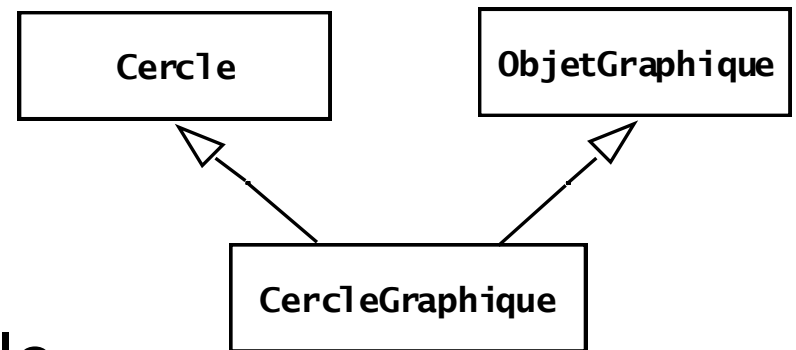
- Notion d'héritage multiple
- Comment définir des comportements communs pour des classes non liées par une relation d'héritage ?
- Comment doter une classe de plusieurs types sans utiliser la relation d'héritage ?
  - Notion d'interface
  - Exemple d'utilisation
- Le qualificateur `final` par rapport à la relation d'héritage

# Rappels sur les propriétés de l'héritage

- Un objet possède plusieurs types :
  - Le type de la classe et les types des super-classes
  - Transtypage explicite parfois nécessaire
- Les avantages :
  - Réduit la duplication de code
  - Favorise la réutilisation de code
- L'inconvénient :
  - Dépendance forte entre les classes liées par cette relation

# Notion d'héritage multiple

- Il peut être naturel d'hériter de plusieurs classes...
- On bénéficie ainsi des méthodes de chaque classe héritée...
- Bien que cela soit possible dans certains langages objet (ex. C++, Python) **ceci ne l'est pas en Java...**



# Notion d'héritage multiple (suite)

- Une des principales raisons est que cela conduit à une plus grande complexité dans l'écriture du compilateur et du code généré
- Cela peut également conduire à des ambiguïtés si les mêmes identificateurs existent dans les classes héritées
- La solution adoptée par Java conduit à hériter d'un type **sans hériter de ses membres...**

# Nécessité de définir des comportements communs sans relation d'héritage

Peut s'assimiler à la liste des méthodes auxquelles l'objet peut répondre

- Supposons qu'on souhaite définir une liste triée pouvant stocker n'importe quel type d'objet...
- Le **comportement** commun de ces objets est de pouvoir se comparer mutuellement :  
`objet1.comparer(objet2)`
- Pour s'assurer que tous les objets insérés dans la liste possèdent ce comportement, il conviendrait de créer une classe abstraite `ElementComparable` déclarant une méthode abstraite `comparer()`

# Nécessité de définir des comportements communs sans relation d'héritage (suite)

- Cela donnerait :

```
public class ListeTrie {  
    // attributs...  
  
    void inserer(EltComparable e) { . . . }  
  
    EltComparable retirerPremier() {  
        return . . . ;  
    }  
}
```

```
abstract class EltComparable {  
    abstract int comparer(Object o);  
}
```

Trois cas possibles codés avec un entier :  
supérieur (+1), inférieur (-1) ou égal (0)

# Nécessité de définir des comportements communs sans relation d'héritage (suite)

```
class Rectangle extends EltComparable {  
    private double surface;  
  
    @Override  
    int comparer(Object o) {  
        if (o == this) return 0;  
        else if (o instanceof Rectangle) {  
            Rectangle r = (Rectangle) o;  
            if (this.surface > r.surface)  
                return +1;  
            else if (this.surface < r.surface)  
                return -1;  
            else  
                return 0;  
        } else throw new IllegalArgumentException();  
    }  
}
```

Comme pour equals, test nécessaire pour s'assurer que l'objet à comparer est du même type...



# Nécessité de définir des comportements communs sans relation d'héritage (suite)

```
public class TestListeTrie {  
    public static void main(String[] args) {  
        Rectangle r = new Rectangle();  
        ListeTrie l = new ListeTrie();  
        l.inserer(r);
```

OK car r est du type `EltComparable`

```
        // malheureusement, si une classe Triangle hérite  
        // de EltComparable, le code dessous se compile  
        Triangle t = new Triangle();  
        l.inserer(t);
```

```
    }
```

```
}
```

# Nécessité de définir des comportements communs sans relation d'héritage (suite)

- Pour éviter le problème, utilisation d'un type paramétrique :

Définition d'une liste « homogène » : elle ne contiendra que des objets du même type T

```
public class ListeTrie<T> {  
    // attributs  
    void inserer(EltComparable<T> e) { . . . }  
  
    EltComparable<T> retirerPremier() { . . . }  
}  
  
abstract class EltComparable<T> {  
    abstract int comparer(T o);  
}
```

# Nécessité de définir des comportements communs sans relation d'héritage (suite)

```
class Rectangle extends EltComparable<Rectangle> {  
    double surface;  
  
    @Override  
    int comparer(Rectangle r) {  
        if (r == this)  
            return 0;  
        else if (this.surface > r.surface)  
            return +1;  
        else if (this.surface < r.surface)  
            return -1;  
        else  
            return 0;  
    }  
}
```

Plus de test car le compilateur pourra vérifier que l'objet passé à la méthode comparer est bien une instance de Rectangle

# Nécessité de définir des comportements communs sans relation d'héritage (suite)

```
public class TestListeTrie {  
    public static void main(String[] args) {  
        Rectangle r = new Rectangle();  
        ListeTrie<Rectangle> l = new ListeTrie<>();  
        l.inserer(r);  
  
        Triangle t = new Triangle();  
        l.inserer(t);  
    }  
}
```

Erreur à la compilation car on ne peut mettre que des rectangles !...

# Nécessité de définir des comportements communs sans relation d'héritage (suite)

- Cette approche a l'**inconconvénient majeur** d'obliger la classe de l'objet contenu dans la liste à hériter de la classe abstraite `ElementComparable`
- Cela l'empêche donc **de pouvoir hériter d'une autre classe** (héritage d'une seule super-classe en Java)
- Exemple : la classe `Cercle` ne pourrait pas être utilisée car elle hérite déjà de `Forme` !...



# Notion d'interface

- Java fournit une solution à ce problème via le mécanisme d'**interface**
- Une interface ressemble à une classe abstraite avec les particularités suivantes :
  - **Toutes** les méthodes sont *implicitement* **abstraites** et **publics**
  - Les seuls attributs autorisés sont des constantes de classe (`static` et `final`)
  - Aucune définition de méthode !...
- Une classe n'hérite pas d'une interface mais **l'implémente...**
- Une classe peut implémenter **plusieurs** interfaces

À partir de Java 8, présence de méthodes implémentées (*default methods*)

# Notion d'interface (suite)

- Reprenons l'exemple précédent :

```
abstract class EltComparable {  
    boolean abstract comparer(EltComparable o);  
}
```



```
interface EltComparable {  
    boolean comparer(EltComparable o);  
}  
  
class Entier extends Nombre  
    implements EltComparable {  
    boolean comparer(EltComparable o) { ... }  
}
```

La classe Entier  
possède maintenant  
3 types :

- Entier
- Nombre
- EltComparable

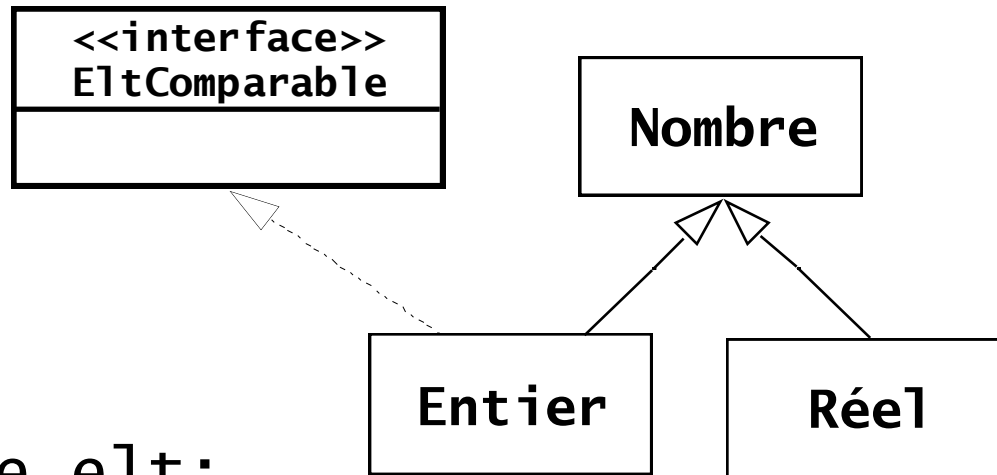
# Notion d'interface (suite)

- Une interface n'est pas une classe :
  - Pas de constructeur et impossible d'utiliser l'opérateur new avec une interface
  - Pas de variables d'instance, ni de classe
  - Pas de définition de méthodes (déclaration uniquement)
- En revanche, une interface est un type donc il est possible :
  - De déclarer des variables dont le type est une interface
  - D'utiliser cette variable pour référencer un objet d'une classe implémentant l'interface
  - Utiliser une interface dans un transtypage



# Notion d'interface (suite)

- Exemple :



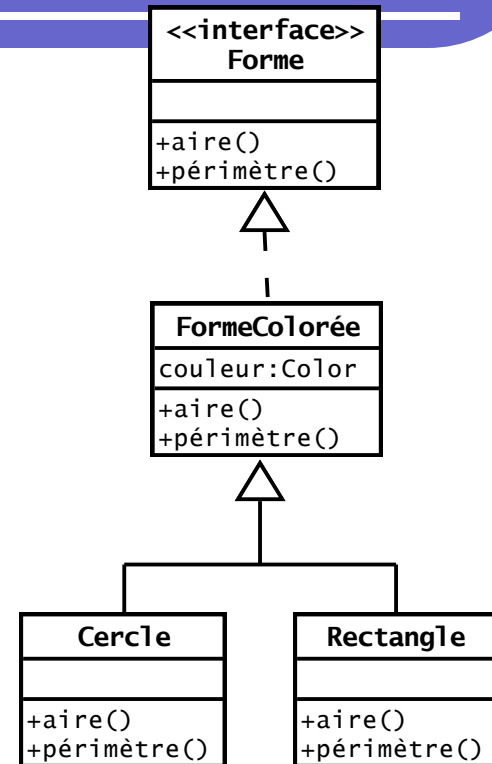
```
EltComparable elt;  
Entier e = new Entier(5);  
elt = e; // OK car Entier implémente EltComparable  
Nombre n = new Entier(6);  
elt = (EltComparable) n;
```

Transtypage nécessaire car tous les objets de type Nombre ne sont pas du type EltComparable...

# Notion d'interface (suite)

- Autre exemple :

Dans le cours précédent  
Forme était présentée  
comme une classe abstraite  
mais elle est mieux  
représentée par une interface  
(dans ce cas, la couleur serait placée dans une  
classe abstraite `FormeColorée` implémentant  
l'interface `Forme`)



# Notion d'interface (suite)

- Une classe peut implémenter plusieurs interfaces :

- Les noms d'interfaces doivent être séparés par des « , »

```
public interface EltPersistant {  
    void        ecrire(Fichier f);  
    EltPersistant lire(Fichier f);  
}  
class Entier extends Nombre  
    implements EltComparable, EltPersistant {  
    boolean comparer(EltComparable o) { ... }  
    void        ecrire(Fichier f)      { ... }  
    EltPersistant lire(Fichier f)      { ... }  
}
```

Définition des méthodes

# Notion d'interface (suite)

- Pour information, il est également possible de définir une hiérarchie d'interfaces :

```
interface EltPersistantComparable  
    extends EltComparable, EltPersistant {  
}
```

Ici extends représente l'union ensembliste ( $\cup$ ) des déclarations de méthodes présentes dans les interfaces

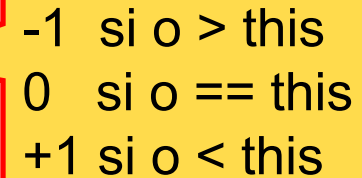
- Les seuls attributs possibles dans la déclaration d'une interface sont des **constantes de classe** :

```
interface EltPersistant {  
    static final int uneConstante = 2;  
    void            ecrire(Fichier f);  
    EltPersistant lire(Fichier f);  
}
```

# Notion d'interface (suite)

- Exemple d'interface dans la bibliothèque Java :

```
interface Comparable<T> {  
    int compareTo(T o);  
}
```



-1 si o > this  
0 si o == this  
+1 si o < this

- Cette interface est importante car elle doit être implémentée par toute classe dont les instances seront stockées dans des structures de données triées tel que TreeSet ou TreeMap...

# Interface et marquage de classes

- Certaines interfaces de la bibliothèque Java ne contiennent **aucune déclaration de méthode**
- Exemples :

```
interface Serializable { }  
interface Cloneable    { }
```
- Ces interfaces vides sont utilisées pour **marquer** (typer) les classes qui les implémentent...



# Interface et marquage de classes (suite)

- Exemple :

```
class A implements Cloneable
{
    public static void main(String[] args) throws Exception {
        A a1 = new A();
        A a2 = (A) a1.clone();
        System.out.println("OK");
    }
}
```

Transtypage  
nécessaire car  
clone retourne  
un Object

```
d:> java A
OK
d:>
```

# Utilisation d'une interface comme type de retour (1)

- Parfois une interface peut être utilisée comme type de retour d'une méthode, cela masque ainsi l'objet réel retourné...
- Exemple :

```
interface Resultat { void m1(); }  
  
class A implements Resultat { void m1() { ... } }  
  
class B implements Resultat { void m1() { ... } }  
  
class Fabrique {  
    public static Resultat retourneAouB(boolean val) {  
        return (val ? new A() : new B());  
    }  
}
```



## Utilisation d'une interface comme type de retour (2)

- Exemple (suite) :

```
class TestResultat {  
    public static void main(String[] args) {  
        Resultat resultat = null;  
        resultat = Fabrique.retourneAouB(true);  
        System.out.println(resultat);  
        resultat = Fabrique.retourneAouB(false);  
        System.out.println(resultat);  
  
        resultat.m1(); // OK  
    }  
}
```

```
> java TestResultat  
A@677327b6  
B@14ae5a5  
>
```

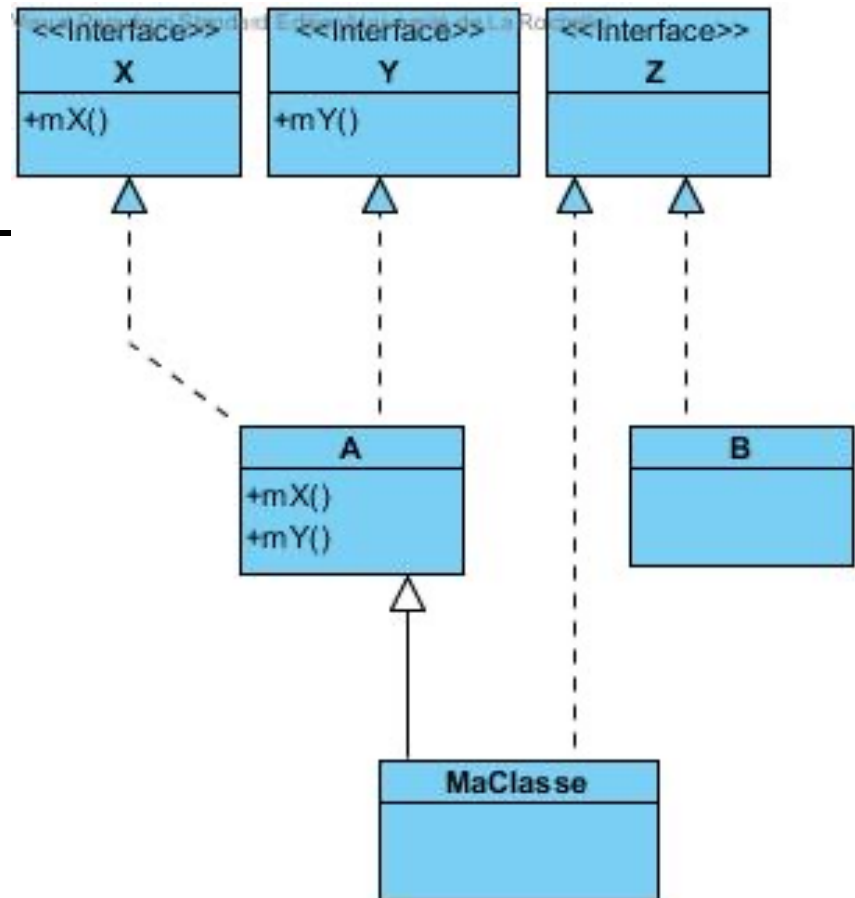
# Notion d'interface (suite)

- En résumé :
  - Une interface est un moyen d'attribuer des **comportements communs** à des classes non liées par une relation d'héritage
  - Chaque classe déclarant implémenter une interface doit définir **toutes les méthodes déclarées** dans cette interface
  - Une classe peut implémenter **autant d'interfaces** qu'elle le souhaite...
  - À la différence d'une classe abstraite, une interface ne fournit **aucune méthode déjà définie**

# Notion d'interface (suite)

- Quizz... Par rapport au diagramme ci-contre, les instructions suivantes sont-elles correctes ?...

```
Y unY = new A();  
Z unZ = new MaClasse();  
Z unAutreZ = new B();  
unZ.mX();  
unZ = unAutreZ;  
A a = unZ;  
unY.mY();  
unY.mX();  
((MaClasse) unY).mX();  
((A) unY).mX();
```



# Exemple d'utilisation

- Avec Java, pour qu'une classe soit réellement utilisable pour instancier des objets graphiques, il est nécessaire qu'elle hérite de `java.awt.Component`
- Si on reprend l'exemple de `CercleGraphique`, elle doit donc hériter de `java.awt.Component`
- Mais elle ne peut alors plus hériter de `Cercle` !  
→ Comment procéder ?...

# Exemple d'utilisation (suite)

- On peut réutiliser une classe avec une relation de composition via une technique appelée **délégation**
- Cela oblige la classe qui « délègue » à implémenter l'ensemble des méthodes de la classe qu'on souhaite réutiliser
- Le code de ces méthodes consiste à appeler les méthodes de l'objet qu'on souhaite réutiliser. Ces méthodes **délèguent** leur travail...

# Exemple d'utilisation (suite)

```
public class CercleGraphique extends java.awt.Component {  
    // réutilisation par délégation de Cercle  
    private Cercle cercle;  
    public double perimetre() { return this.cercle.perimetre(); }  
    public double aire()      { return this.cercle.aire();}  
    // partie spécifique à CercleGraphique  
    private java.awt.Color couleur;  
    public CercleGraphique(double r, Point o, java.awt.Color c) {  
        this.cercle = new Cercle(r,o);  
        this.couleur = c;  
    }  
    public void paint(java.awt.Graphics g) {  
        g.setColor(couleur);  
        g.drawOval(this.cercle.donneCentre().donneX(),...);  
    }  
}
```

Les appels sont  
délégués à l'instance  
de Cercle

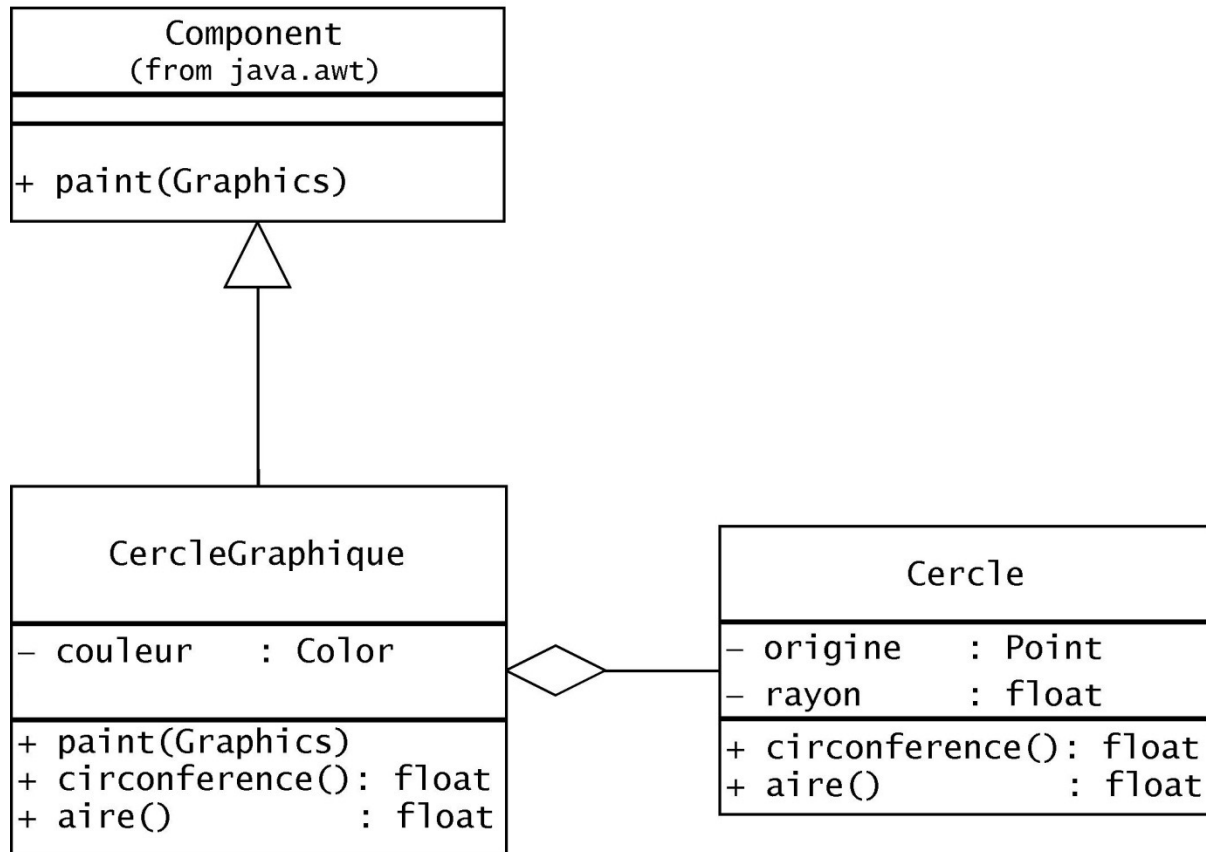
# Exemple d'utilisation (suite)

- Cette transformation permet à la classe `CercleGraphique` de pouvoir répondre aux appels de méthodes de la classe `Cercle`, elle possède donc *le même comportement*
- Cependant, cette approche présente plusieurs inconvénients :
  - Il est nécessaire de redéfinir toutes les méthodes qui étaient auparavant héritées !...
  - **Plus grave**, une instance de `CercleGraphique` ne peut plus être considérée comme une instance de `Cercle`



# Exemple d'utilisation (suite)

- La situation actuelle...



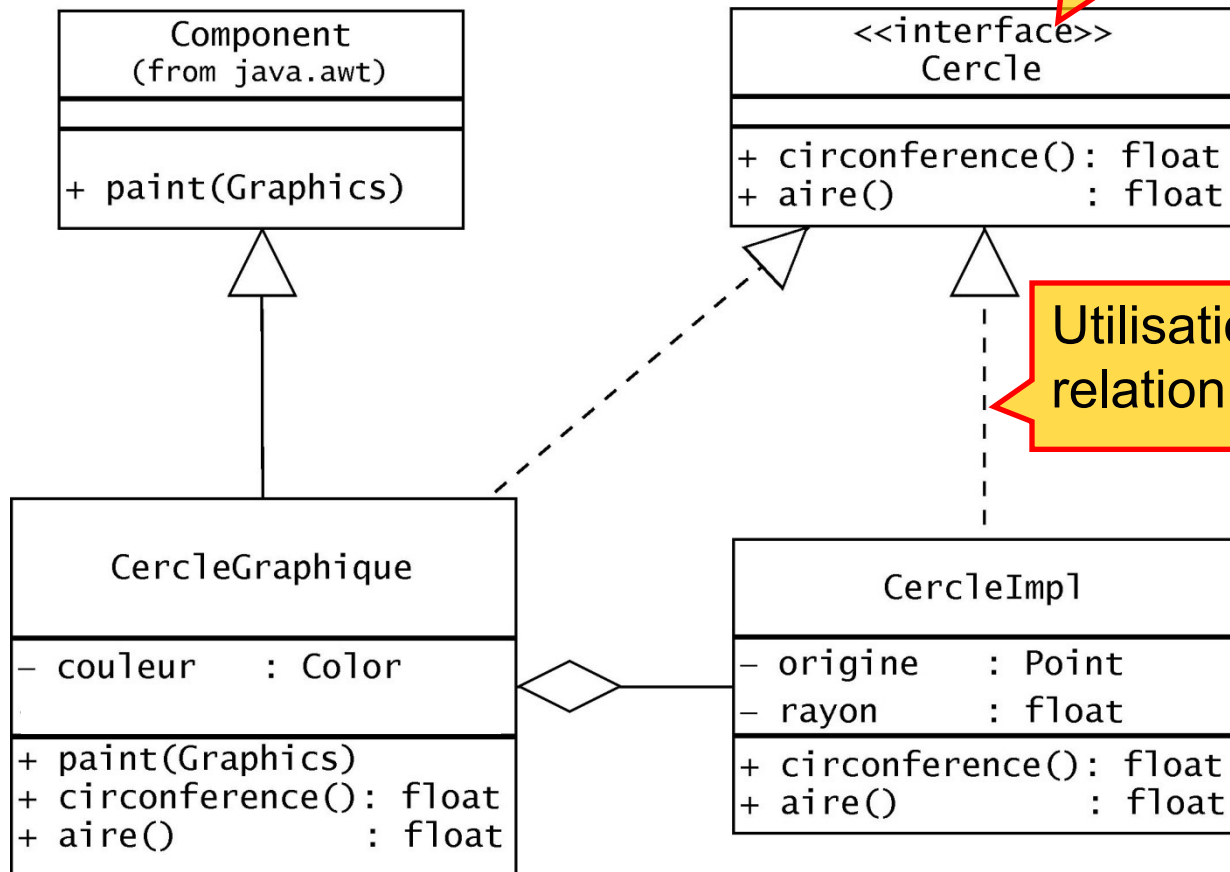


# Exemple d'utilisation (suite)

- Comment procéder pour conserver le type `Cercle` pour la classe `CercleGraphique` ?...
  - La solution est de transformer la classe `Cercle` en **interface**...
  - Comme `CercleGraphique` implémentera `Cercle`, elle possédera également le type `Cercle`...
  - La classe `Cercle` originale deviendra une classe `CercleImpl` qui implémentera également `Cercle`

# Exemple d'utilisation (suite)

- La solution...



Création d'une interface Cercle

Utilisation de la relation implements

# Exemple d'utilisation (suite)

- Limites de cette technique :
  - Il est nécessaire **de posséder le code source** d'une des deux classes dont on souhaite hériter (ici c'était le cas pour Cercle)
  - Dans le cas contraire, (ex. deux classes de la bibliothèque Java) cela n'est pas possible
    - Cependant la bibliothèque Java anticipe ce problème en offrant, dans de nombreux cas, à la fois une classe et l'interface de cette classe...
    - Exemple : `WindowListener` et `WindowAdapter`

# Exemple d'utilisation (suite et fin)

```
interface WindowAdapter {  
    void windowClosed(WindowEvent e);  
    void windowOpened(WindowEvent e);  
    ... // 10 méthodes !...  
}  
  
public class WindowListener implements WindowAdapter  
{  
    void windowClosed(WindowEvent e) { }  
    void windowOpened(WindowEvent e) { }  
    ... // par défaut, les 10 méthodes ne font rien...  
        // mais elles sont définies ! 😊  
}
```

# Le qualificateur `final`

- Le qualificateur `final` s'applique :
  - Aux variables et aux paramètres (cf. cours I)
  - Mais, également, **aux méthodes et aux classes...**
- Sa sémantique est toujours d'**interdire les modifications**
  - Qu'est-ce que cela signifie pour les méthodes et les classes ?...

# Le qualificateur `final` (suite)

- Si on qualifie `final` une méthode, cela **interdit sa redéfinition** dans une sous-classe...
- Exemple :

```
public class Rectangle extends Forme {  
    ...  
    public final double aire() {  
        return this.long * this.larg;  
    }  
    public final double perimetre() {  
        return 2*(this.long + this.larg);  
    }  
}
```

On ne souhaite pas que ces méthodes soient redéfinies dans des sous-classes de `Rectangle`...

# Le qualificateur `final` (suite)

- Le qualificateur `final` appliqué à une méthode :
  - Empêche la redéfinition de la méthode
  - Et donc annule la résolution dynamique (c'est-à-dire le choix de la méthode à l'exécution)

```
class A {  
    final void m() { }  
}
```

```
class B extends A {  
}
```

```
A a = new B();  
a.m();
```

Ici le compilateur n'a pas besoin de générer un test pour déterminer le type dynamique car la méthode `m` ne peut pas exister dans une sous-classe de `A`

- Utilisé dans la bibliothèque Java pour empêcher que le code de certaines méthodes soit modifié en les redéfinissant (pour des problèmes de sécurité notamment)

# Le qualificateur `final` (suite)

- Le qualificateur `final` peut également être appliqué à une classe :
  - Cela empêche la création de sous-classes
  - Cela permet de **fixer de manière définitive** le comportement d'une classe
- Exemple : la classe `String`

```
public final class String  
    extends Object  
    implements Serializable, Comparable<String>, CharSequence
```



# Pour résumer...

- La notion d'interface permet d'éviter l'emploi de la relation d'héritage lorsque ce n'est pas possible ou lorsque ce n'est pas souhaitable (imposer uniquement un comportement)
- Il est possible en cas d'héritage multiple de transformer des classes en interfaces (à condition d'avoir accès au code source) pour revenir à de l'héritage simple