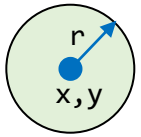
# Héritage et abstraction
## classes abstraites

Philippe Genoud

# Classes Abstraites

## Exemple introductif

- un **grand classique :** les formes géométriques
  - on veut définir une application permettant de manipuler des formes géométriques (triangles, rectangles, cercles…).
  - chaque forme est définie par sa position dans le plan
  - chaque forme peut être déplacée (modification de sa position), peut calculer son périmètre, sa surface

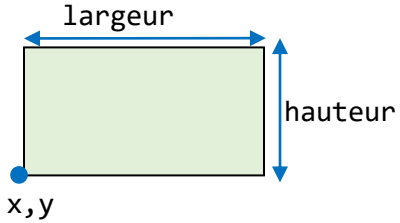



**Attributs :**

```
double x,y; //centre du cercle
double r; // rayon
```

**Méthodes :**

```
deplacer(double dx, double dy)
double surface()
double périmètre()
```

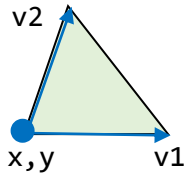



**Attributs :**

```
double x,y; //coin inférieur gauche
double largeur, hauteur;
```

**Méthodes :**

```
deplacer(double dx, double dy)
double surface()
double périmètre();
```




**Attributs :**

```
double x,y; // un des sommets
double x1,y1; // v1
double x2,y2; // v2
```

**Méthodes :**

```
deplacer(double dx, double dy)
double surface()
double périmètre();
```

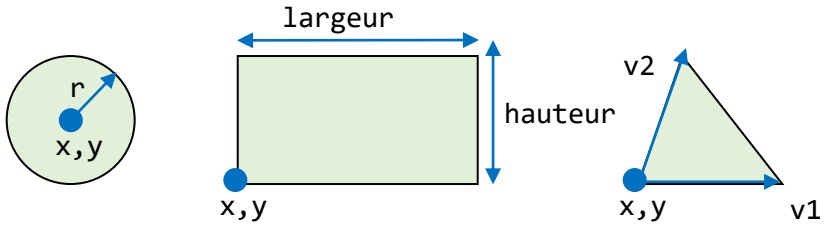
Factoriser le code ?

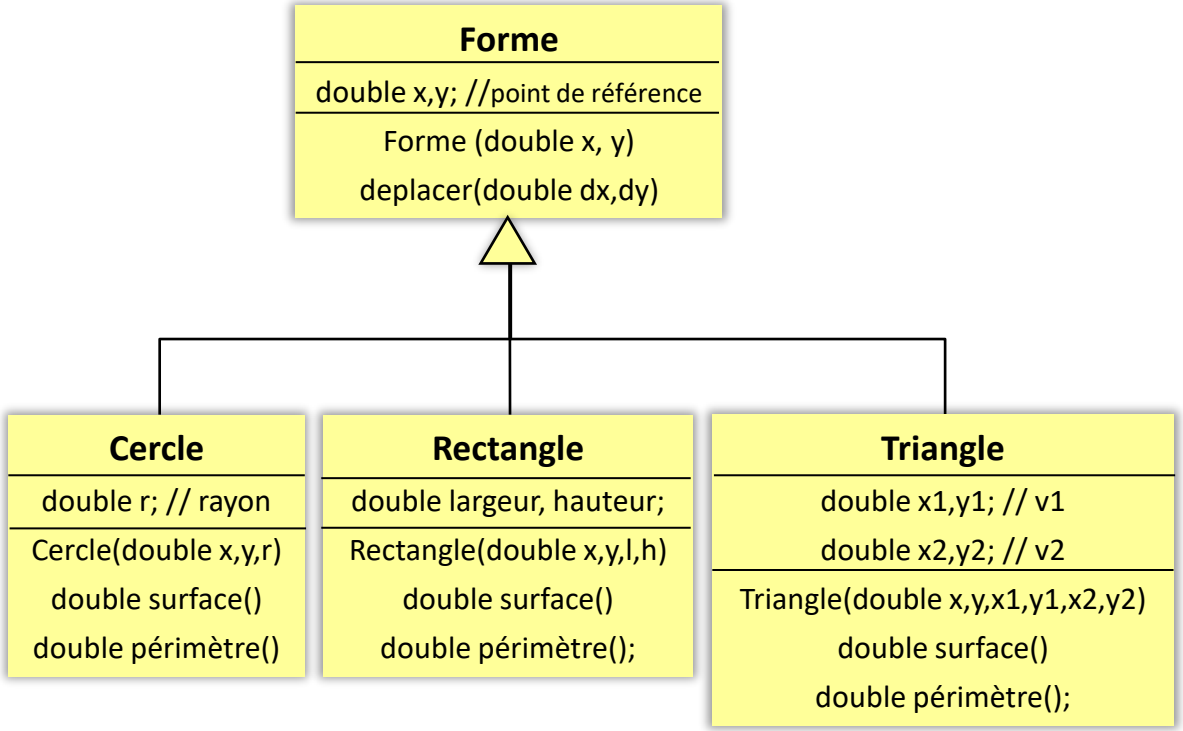
# Classes abstraites

## Exemple introductif

```
public class Forme {
   protected double x,y;

   public Forme(double x, double y) {
     this.x = x;
     this.y = y;
   }

   public void deplacer(double dx,
       double dy) {
      x += dx; y += dy;
   }
}
```

```
public class Cercle extends Forme {
    protected double r;

    public Cercle(double x, double y,
                  double r) {
       super(x,y);
       this.r = r;
     }

    public double surface(){
       return Math.PI * r * r;
    }

    protected double périmetre(){
       return 2 * Math.PI * r;
    }
}
```



# Classes abstraites

## Exemple introductif

```
public class ListeDeFormes {

  public static final int NB_MAX = 30;
  private Forme[] tabFormes = new Forme[NB_MAX];
  private int nbFormes = 0;

  public void ajouter(Forme f) {
    if (nbFormes < NB_MAX)
       tabFormes[nbFormes++] =f
  }

  public void toutDeplacer(double dx,double dy) {
    for (int i=0; i < nbFormes; i++)
       tabFormes[i].deplacer(dx,dy);
  }


  public double périmètreTotal() {
    double perimTotal = 0.0;
    for (int i=0; i < nbFormes++; i++)
         perimTotal += tabFormes[i].périmètre();
    return perimTotal;
  }

}
```


On veut pouvoir gérer des listes de formes

On exploite le **polymorphisme**
la prise en compte de nouveaux types de forme ne modifie pas le code

Appel non valide car la méthode **`périmètre`** n'est pas implémentée au niveau de la classe **`Forme`**

Définir une méthode **`périmètre`** dans **`Forme`** ?

```
public double périmètre() {
  return 0.0; // ou -1. ??
}
```


Une solution propre et élégante : les **classes abstraites**

# Classes abstraites

Classe abstraite

```
public abstract class Forme {
   protected double x,y;

     public Forme(double x, double y) {
       this.x = x;
       this.y = y;
     }

   public void deplacer(double dx,
         double dy) {
     x += dx; y += dy;
   }
    public abstract double périmètre();

    public abstract double surface();

  }
```

la classe doit être déclarée comme étant explicitement abstraite

Méthodes abstraites

on spécifie qu'un objet de type Forme aura une méthode périmètre et une méthode surface

par contre on ne sait pas comment cela sera implémenté → méthodes sans corps :
; au lieu de { … }

- Utilité :
  - définir des concepts incomplets qui devront être implémentés dans les sous classes
  - factoriser le code
  - les opérations abstraites sont particulièrement utiles pour mettre en œuvre le polymorphisme.
    - l'utilisation du nom d'une classe abstraite comme type pour une (des) référence(s) est toujours possible (et souvent souhaitable !!!)

# Classes abstraites

- **classe abstraite** : classe non instanciable, c'est à dire qu'elle n'admet pas d'instances directes.
  - impossible de faire `new ClasseAbstraite(…);`
  - mais une classe abstraite peut néanmoins avoir un ou des constructeurs

- **opération abstraite** : opération n'admettant pas d'implémentation
  - au niveau de la classe dans laquelle elle est déclarée, on ne peut pas dire comment la réaliser.

- Une classe pour laquelle au moins une opération abstraite est déclarée est nécessairement une classe abstraite (l'inverse n'est pas vrai).

```
public abstract class ClasseA {

   ...

   public abstract void methodeA();

   ...
}
```

la classe contient une méthode abstraite => elle **doit** être déclarée abstraite

```
public abstract  class ClasseA {

   ...



}
```

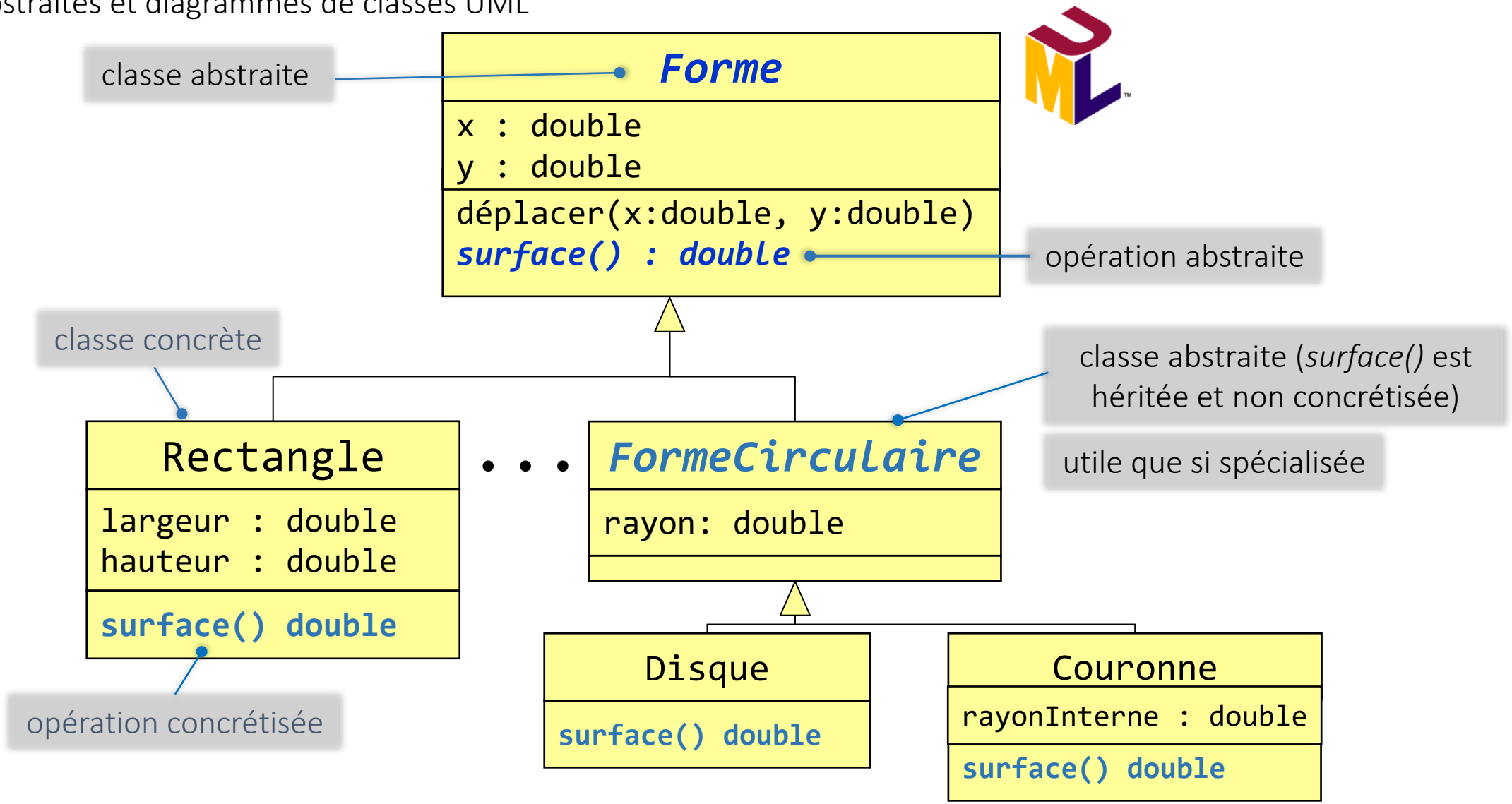
la classe ne contient pas de méthode abstraite
=> elle **peut** néanmoins être déclarée abstraite

# Classes abstraites

- Une classe abstraite est une description d'objets destinée à être héritée par des classes plus spécialisées.
- Pour être utile, une classe abstraite doit admettre des classes descendantes ***concrètes***.
- Toute classe **concrète** sous-classe d'une classe abstraite doit “concrétiser” toutes les opérations abstraites de cette dernière.
  - elle doit implémenter toutes les méthodes abstraites
- Une classe abstraite permet de regrouper certaines caractéristiques communes à ses sous-classes et définit un comportement minimal commun.
- La factorisation optimale des propriétés communes à plusieurs classes par généralisation nécessite souvent l'utilisation de classes abstraites.

# Classes abstraites

Classes abstraites et diagrammes de classes UML



# Classes abstraites

## Exemple introductif

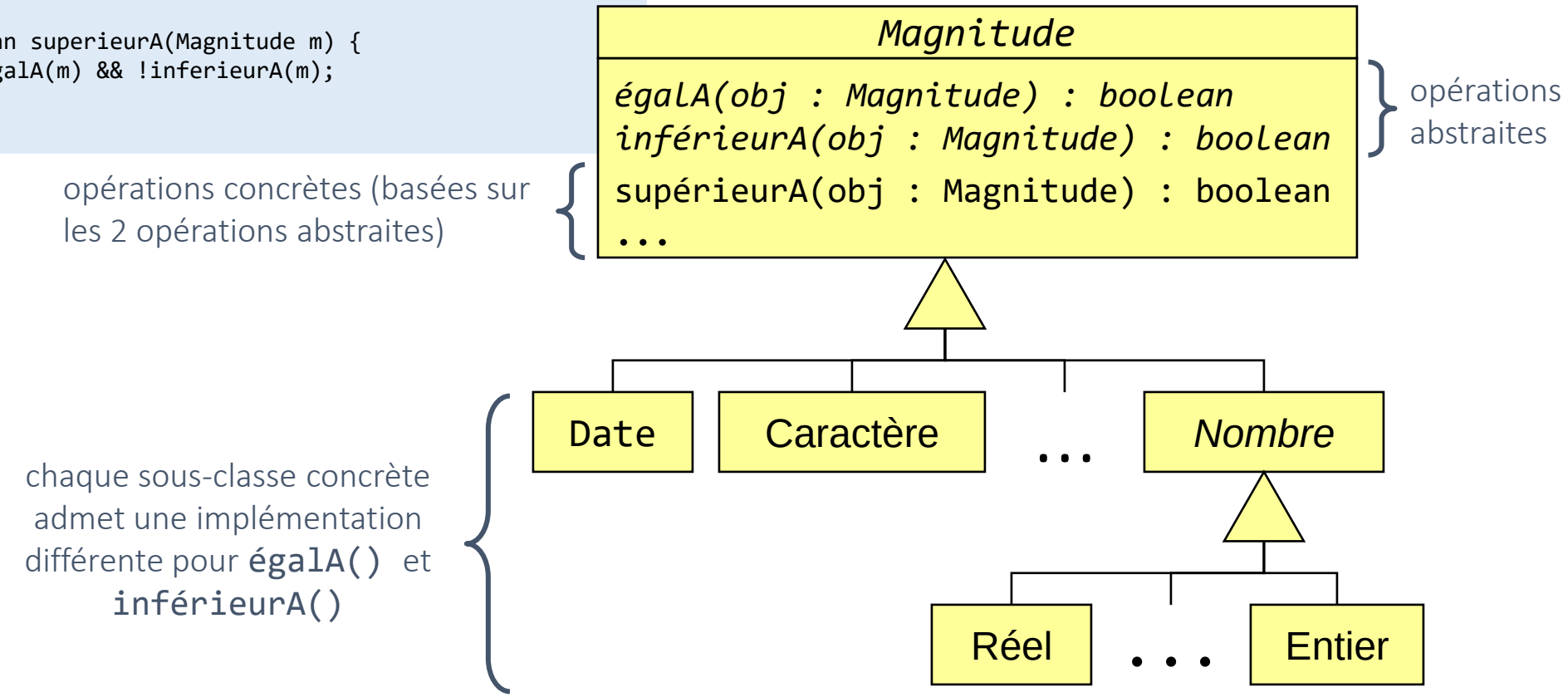
```
public class ListeDeFormes {

  public static final int NB_MAX = 30;
  private Forme[] tabFormes = new Forme[NB_MAX];
  private int nbFormes = 0;

  public void ajouter(Forme f) {
    if (nbFormes < NB_MAX)
       tabFormes[nbFormes++] =f
  }

  public void toutDeplacer(double dx,double dy) {
    for (int i=0; i < NbFormes; i++)
       tabFormes[i].deplacer(dx,dy);
  }


  public double périmètreTotal() {
    double perimTotal = 0.0;
    for (int i=0; i < nbFormes++; i++)
         perimTotal += tabFormes[i].périmètre();
    return perimTotal;
  }

}
```

```
public abstract class Forme {
  protected double x,y;

  public Forme(double x, double y) {
      this.x = x;
      this.y = y;
  }

  public void deplacer(double dx,
       double dy) {
     x += dx; y += dy;
  }

  public abstract double périmètre();
  public abstract double surface();
}
```

Le polymorphisme peut être pleinement exploité.
Le compilateur sait que chaque objet **Forme** peut calculer son périmètre

# Classes abstraites

```
public abstract class Magnitude {

   public abstract boolean egalA(Magnitude m) ;

   public abstract boolean inferieurA(Magnitude m) ;

   public boolean superieurA(Magnitude m) {
      return !egalA(m) && !inferieurA(m);
   }
   ...
```




**(exemple inspiré du cours GL de D. Bardou, UPMF)**

# Héritage et abstraction
## interfaces

Philippe Genoud

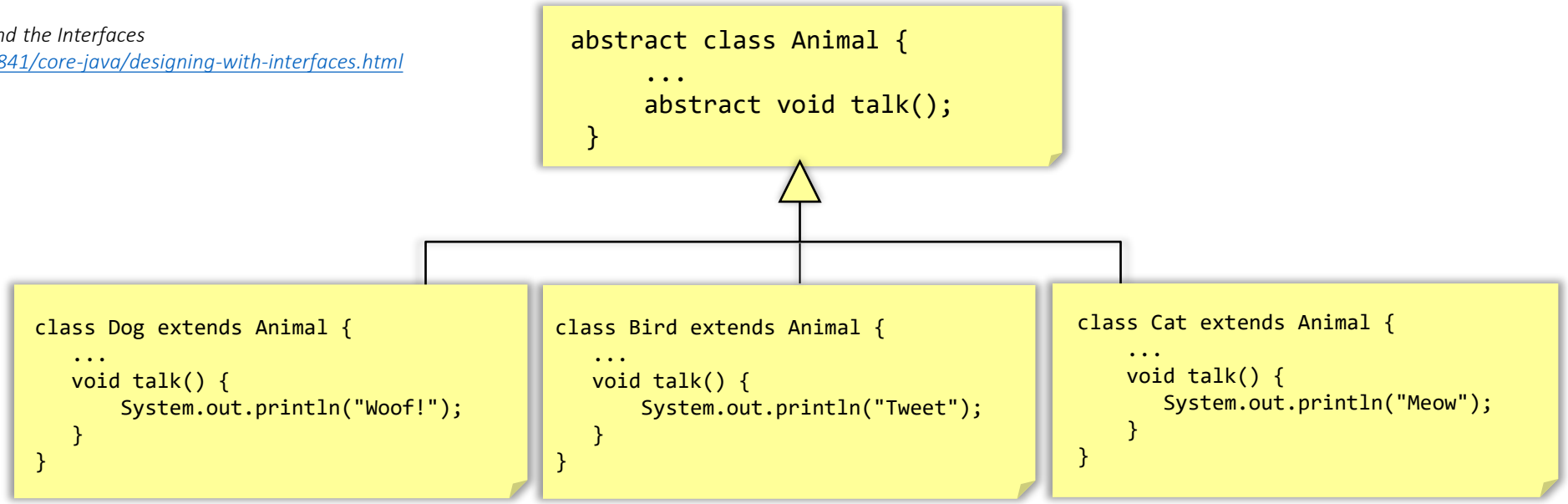
# Interfaces

## Exemple introductif

Bill Venners *Designing with Interfaces*

*One Programmer's Struggle to Understand the Interfaces*

*http://www.javaworld.com/article/2076841/core-java/designing-with-interfaces.html*




**Polymorphisme** signifie qu'une référence d'un type (classe) donné peut désigner un objet de n'importe quelle sous classe et selon la nature de cet objet produire un comportement différent

En JAVA le polymorphisme est rendu possible par la **liaison dynamique** (*dynamic binding*)

JVM **décide à l'exécution** (runtime) quelle méthode invoquer en se basant sur la classe de l'objet

```
class Interrogator {

    static void makeItTalk(Animal subject) {
        subject.talk();
    }
}
```

`animal` **peut être** un `Chien`, un `Chat` ou n'importe quelle sous classe d'`Animal`

```
Dog pluto = new Dog();
Cat fritz = new Cat();
Interrogator.makeItTalk(pluto);
Interrogator.makeItTalk(fritz);
Bird titi = new Bird();
Interrogator.makeItTalk(titi);
```

→ `Woof!`

→ `Meow`

→ `Tweet`

## Exemple introductif

Comment utiliser `Interrogator` pour faire parler aussi un `CuckooClock` ?

```
abstract class Animal {

     abstract void talk();
 }
```

Faire rentrer `CuckooClock` dans la hiérarchie Animal ?

```
class Clock {

    ...
}
```

Pas d'héritage multiple

```
class Dog extends Animal {

    void talk() {
        System.out.
    }
}
```

```
class Bird extends Animal {

     void talk() {
        System.out
     }
}
```

```
class Cat extends Animal {

     void talk() {
        System.out.println("Meow.");
     }
}
```

```
class CuckooClock extends Clock {
  public void talk() {
     System.out.println("Cuckoo,
           cuckoo!");
  }
}
```

```
class Interrogator {

     static void makeItTalk(Animal subject) {
         subject.talk();
     }
 }
```

```
class CuckooClockInterrogator {

     static void makeItTalk(Cuckooclock subject) {
         subject.talk();
     }
 }
```

Se passer du polymorphisme ?

## Exemple introductif

Comment utiliser `Interrogator` pour faire parler aussi un `CuckooClock` ?

```
abstract class Animal implements Talkative {

    abstract void talk();
}
```

```
class Dog extends Animal {

    void talk() {
        System.out.
    }
}
```

```
class Bird extends Animal {

    void talk() {
        System.ou
    }
}
```

```
class Cat extends Animal {

    void talk() {
        System.out.println("Meow.");
    }
}
```

Association de ce type à différentes classes de la hiérarchie d'héritage

```
class Clock {

    ...
}
```

```
class CuckooClock extends Clock implements Talkative {
  public void talk() {
     System.out.println("Cuckoo,
           cuckoo!");
  }
}
```

```
class Interrogator {

    static void makeItTalk(Talkative subject) {
        subject.talk();
    }
}
```

Utilisation de ce type abstrait

```
interface Talkative {

    public void talk();
}
```

Définition d'un type abstrait (interface)

- Les interfaces permettent **plus de polymorphisme** car avec les interfaces il n'est pas nécessaire de tout faire rentrer dans une seule famille (hiérarchie) de classes

# Interfaces

*« Java's interface gives you more polymorphism than you can get with singly inherited families of classes, without the "burden" of multiple inheritance of implementation. »*

Bill Venners *Designing with Interfaces –*
*One Programmer's Struggle to Understand the Interface*
http://www.atrima.com/designtechniques/index.html

# Interfaces

## déclaration d'une interface

- Une *interface* est une collection d'opérations utilisée pour spécifier un service offert par une classe.
- Une interface peut être vue comme une classe 100% abstraite sans attributs et dont toutes les opérations sont abstraites.

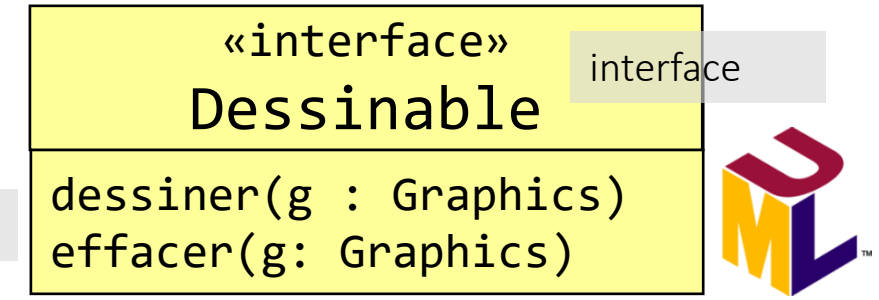
Une interface non publique n'est accessible que dans son package

```
package m2pcci.dessin;
import java.awt.Graphics;

public interface Dessinable {
   public void dessiner(Graphics g);
   void effacer(Graphics g);
}
```

Possibilité d'implémen-tation par défaut avec Java 8

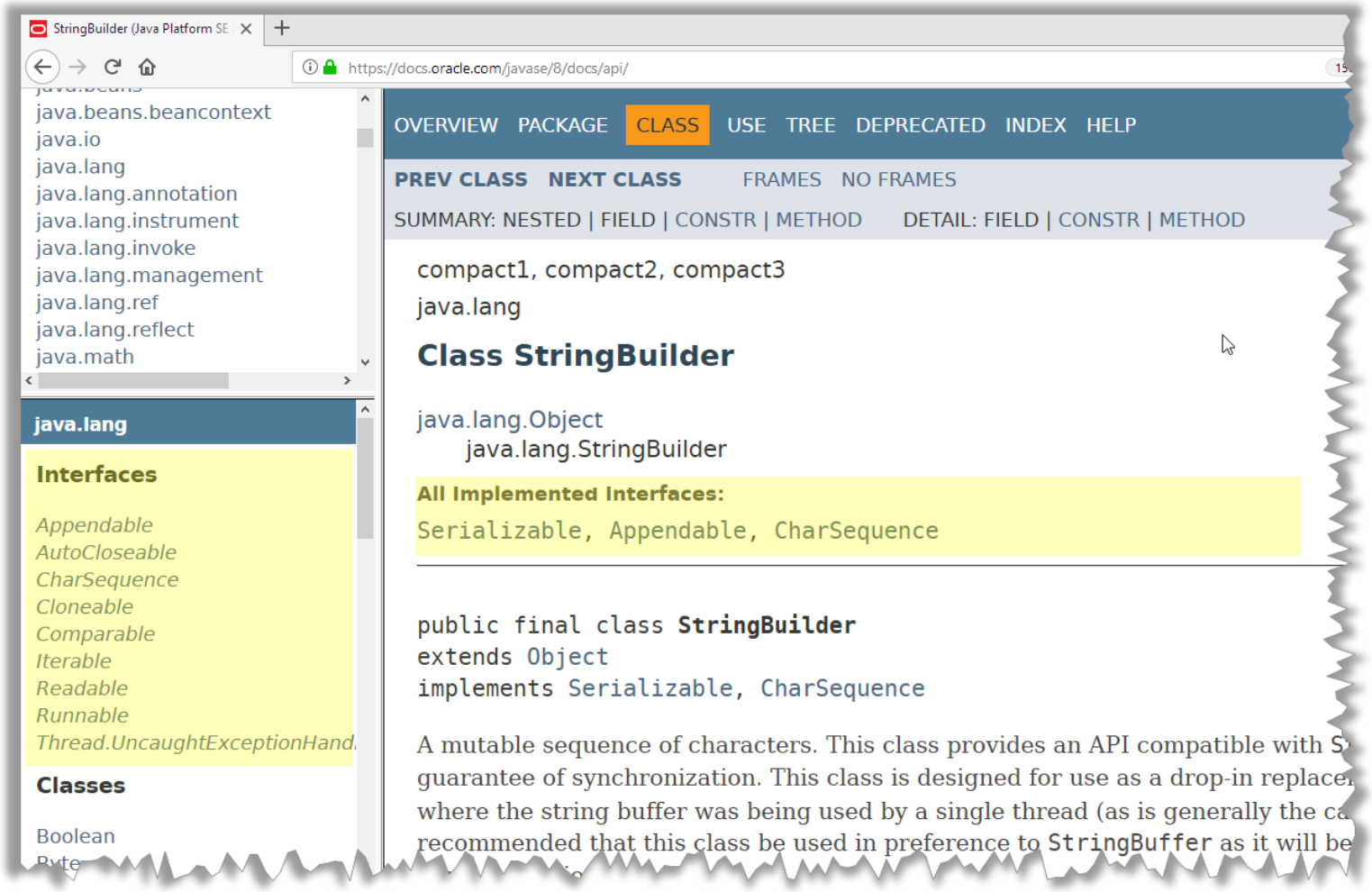
Toutes les méthodes sont abstraites
Elles sont implicitement publiques


Une interface publique doit être définie dans un fichier `.java` de même nom **`Dessinable.java`**



# Interfaces dans la javadoc

https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html




Interfaces du package

Interfaces implémentées par la classe

## déclaration d'une interface

- Possibilité de définir des attributs
- Ces attributs sont implicitement déclarés comme `static final`

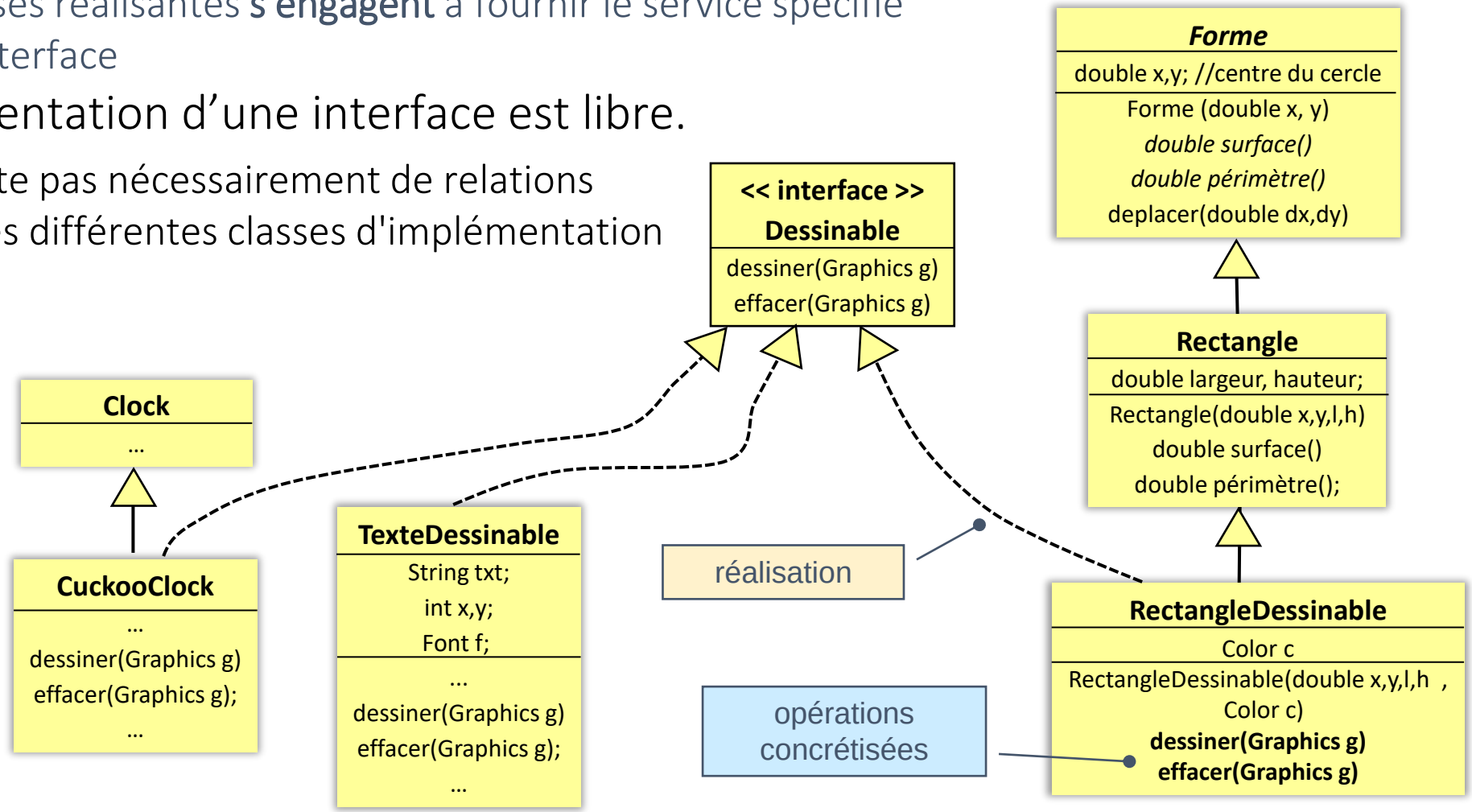
```
import java.awt.Graphics;
public interface Dessinable {

   public static final int MAX_WIDTH = 1920;

   int MAX_HEIGHT = 1080;

    public void dessiner(Graphics g);
    void effacer(Graphics g);

 }
```

**Dessinable.java**

# Interfaces

## "réalisation" d'une interface

- Une interface est destinée à être *"réalisée"* (*implémentée*) par d'autres classes (celles-ci en héritent toutes les descriptions et concrétisent les opérations abstraites).
  - les classes réalisantes **s'engagent** à fournir le service spécifié par l'interface
- L'implémentation d'une interface est libre.
  - il n'existe pas nécessairement de relations entre les différentes classes d'implémentation



# Interfaces

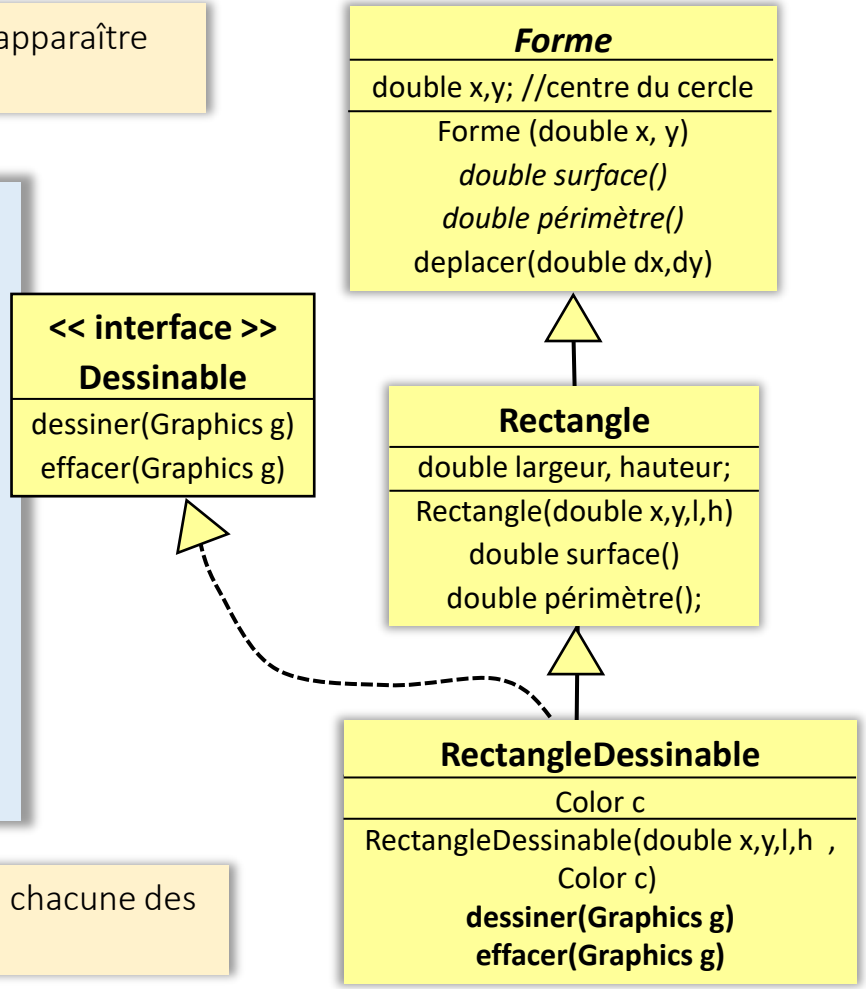
## "réalisation" d'une interface

- De la même manière qu'une classe étend sa super-classe elle peut de manière **optionnelle** implémenter une ou plusieurs interfaces

dans la définition de la classe, après la clause `extends nomSuperClasse`, faire apparaître explicitement le mot clé `implements` suivi du nom de l'interface implémentée

```
class RectangleDessinable extends Rectangle implements Dessinable {
    private Color c;

    public RectangleDessinable(double x, double y,
                double l, double h, Color c) {
         super(x,y,l,h);
         this.c = c;
    }

    public void dessiner(Graphics g){
      g.drawRect((int) x, (int) y, (int) largeur, (int) hauteur);
    }
    public void effacer(Graphics g){
      g.clearRect((int) x, (int) y, (int)largeur, (int) hauteur);
    }
}
```

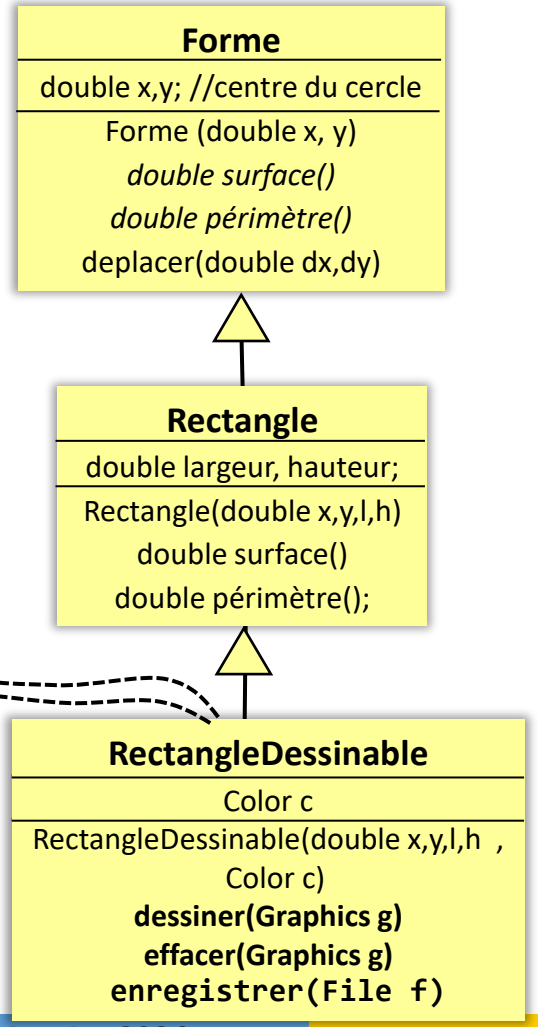
si la classe est une classe concrète elle doit fournir une implémentation (un corps) à chacune des méthodes abstraites définies dans l'interface (qui doivent être déclarées publiques)



- Une classe JAVA peut implémenter **simultanément plusieurs** interfaces

la liste des noms des interfaces à implémenter séparés par des virgules doit suivre le mot clé **`implements`**

```
class RectangleDessinable extends Rectangle
                          implements Dessinable , Enregistrable {
   private Color c;

    public RectangleDessinable(double x, double y,
               double l, double h, Color c) {
         super(x,y,l,h);
         this.c = c;
    }
    public void dessiner(Graphics g){
      g.drawRect((int) x, (int) y, (int) largeur, (int) hauteur);
    }
    public void effacer(Graphics g){
      g.clearRect((int) x, (int) y, (int)largeur, (int) hauteur);
    }

    public void enregistrer(File f) {
      ...
    }
}
```

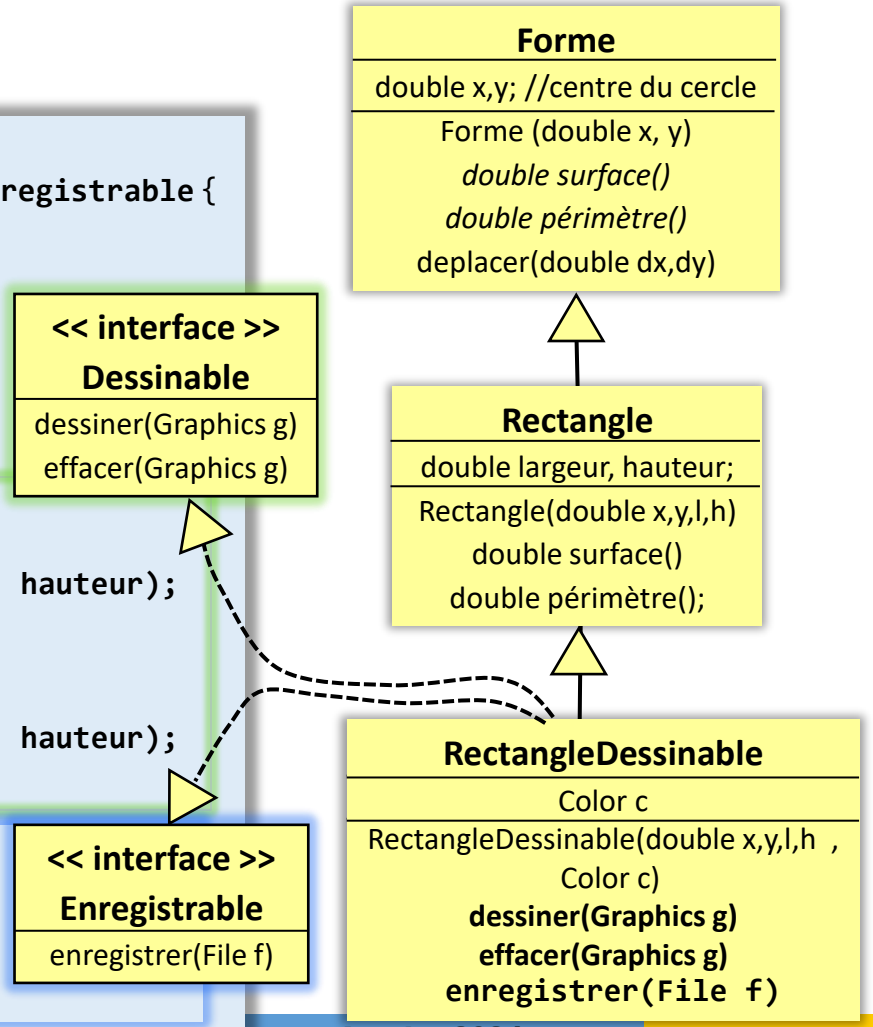


# Interfaces

## "réalisation" d'une interface

- pour éviter des surcharges plutôt que des rdéfinitions de méthodes penser à mettre des directives `@Override` lorsque implémentation des méthodes d'une interface

```
class RectangleDessinable extends Rectangle
                              implements Dessinable , Enregistrable {
   private Color c;

    public RectangleDessinable(double x, double y,
                double l, double h, Color c) {
          super(x,y,l,h);
          this.c = c;
    }
    @Override
    public void dessiner(Graphics g){
      g.drawRect((int) x, (int) y, (int) largeur, (int) hauteur);
    }
    @Override
    public void effacer(Graphics g){
      g.clearRect((int) x, (int) y, (int)largeur, (int) hauteur);
    }

    @Override
    public void enregistrer(File f) {
      ...
    }

}
```



- Une interface peut être utilisée comme un type
  - A des variables (références) dont le type est une interface il est possible d'affecter des instances de toute classe implémentant l'interface, ou toute sous-classe d'une telle classe.

```
public class ZoneDeDessin {
    private nbFigures;
    private Dessinable[] figures;
    ...
    public void ajouter(Dessinable d){
      ...
    }
    public void supprimer(Dessinable o){
      ...
    }

    public void dessiner() {
         for (int i = 0; i < nbFigures; i++)
             figures[i].dessiner(g);
    }

}
```

```
Dessinable d;
..
d = new RectangleDessinable(…);
...
d.dessiner(g);
d.surface();
```

Erreur de compilation, la méthode surface n'est pas définie dans le type **Dessinable**

permet de s'intéresser uniquement à certaines caractéristiques d'un objet

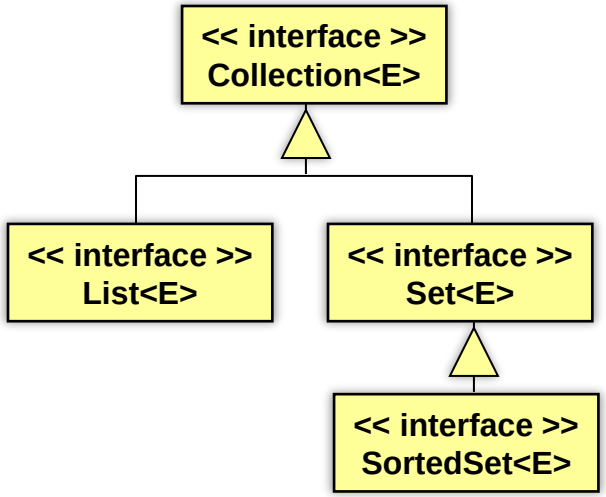
règles du polymorphisme s'appliquent de la même manière que pour les classes :

- vérification statique du code
- liaison dynamique

# Interfaces

## héritage d'interfaces

- De la même manière qu'une classe peut avoir des sous-classes, une interface peut avoir des "sous-interfaces"
- Une sous interface
  - hérite de toutes les méthodes abstraites et des constantes de sa "super-interface"
  - peut définir de nouvelles constantes et méthodes abstraites



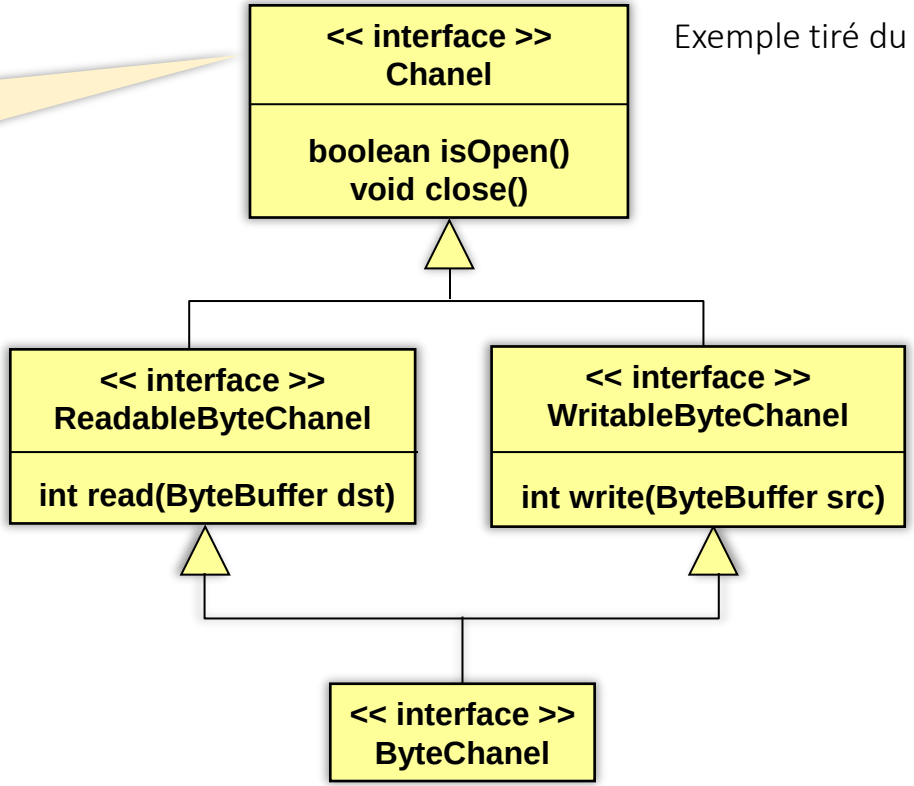

??? Types génériques (ou types paramétrés).
On verra cela dans un cours ultérieur.

```
interface Set<E> extends Collection<E> {
   ...

}
```

- Une classe qui implémente une interface doit implémenter toutes les méthodes abstraites définies dans l'interface et dans les interfaces dont elle hérite.

- A la différence des classes une interface peut étendre plus d'une interface à la fois (héritage multiple sur les interfaces)

représente une connexion ouverte vers une entité telle qu'un dispositif hardware, un fichier, une "socket" réseau, ou tout composant logiciel capable de réaliser une ou plusieurs opérations d'entrée/sortie.




Exemple tiré du package `java.nio`

```
package java.nio;
interface ByteChannel extends ReadableByteChanel, WriteableByteChanel {
}
```

- Les interfaces permettent de s ’affranchir d ’éventuelles contraintes d’héritage.
  - Lorsqu’on examine une classe implémentant une ou plusieurs interfaces, on est sûr que le code d’implémentation est dans le corps de la classe. Excellente localisation du code (défaut de l’héritage multiple, sauf si on hérite de classes purement abstraites).

- Permet une **grande évolutivité** du modèle objet

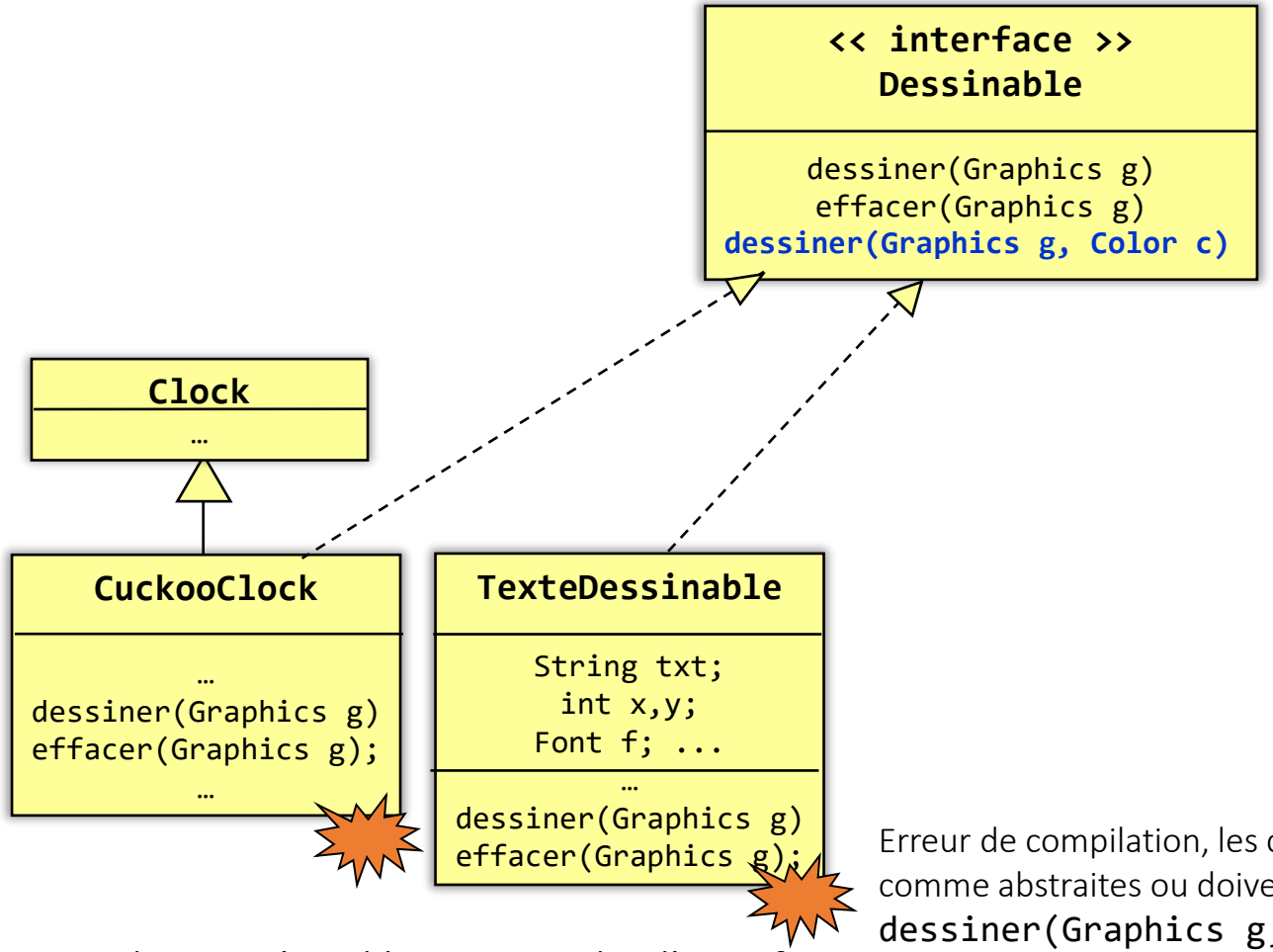
Smarter Java development - *Michael Cymerman* , javaworld août 99. http://www.javaworld.com

*« By incorporating interfaces into your next project, you will notice benefits throughout the lifecycle of your development effort. The technique of coding to interfaces rather than objects will improve the efficiency of the development team by:*

- *Allowing the development team to quickly establish the interactions among the necessary objects, without forcing the early definition of the supporting objects*
- *Enabling developers to concentrate on their development tasks with the knowledge that integration has already been taken into account*
- *Providing flexibility so that new implementations of the interfaces can be added into the existing system without major code modification*
- *Enforcing the contracts agreed upon by members of the development team to ensure that all objects are interacting as designed »*

# Evolution des interfaces

- Quand on définit des interfaces il faut être prudent : tout ajout ultérieur "brise" le code des classes qui implémentent l'interface.

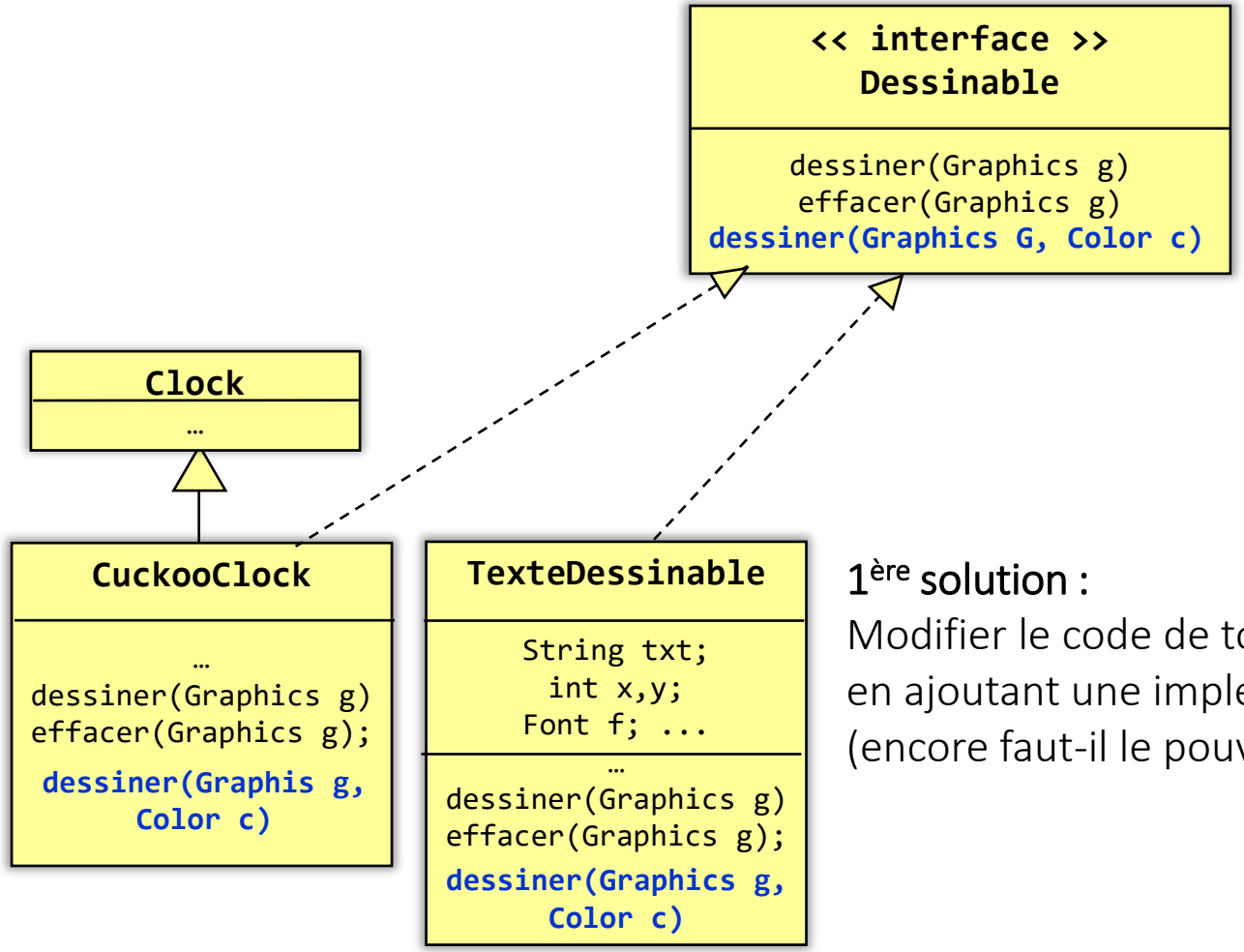



Ces classes n'implémentent plus l'interface

Erreur de compilation, les classes doivent être déclarées comme abstraites ou doivent implémenter la méthode `dessiner(Graphics g, Color c)`

# Evolution des interfaces

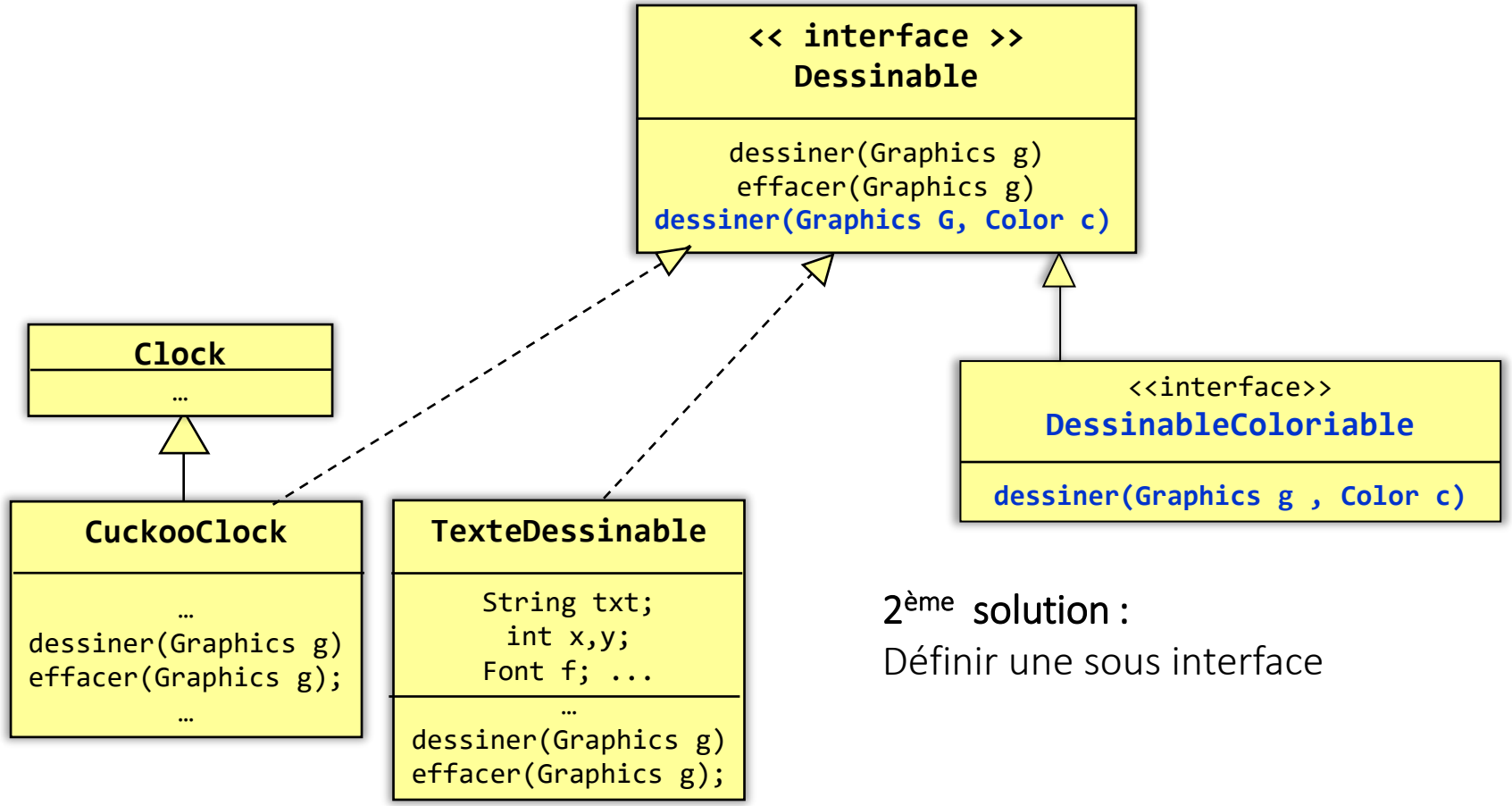
- Quand on définit des interfaces il faut être prudent : tout ajout ultérieur "brise" le code des classes qui implémentent l'interface.




1ère solution :
Modifier le code de toutes les classes implémentant l'interface en ajoutant une implémentation pour la nouvelle méthode (encore faut-il le pouvoir)

# Evolution des interfaces

- Quand on définit des interfaces il faut être prudent : tout ajout ultérieur "brise" le code des classes qui implémentent l'interface.

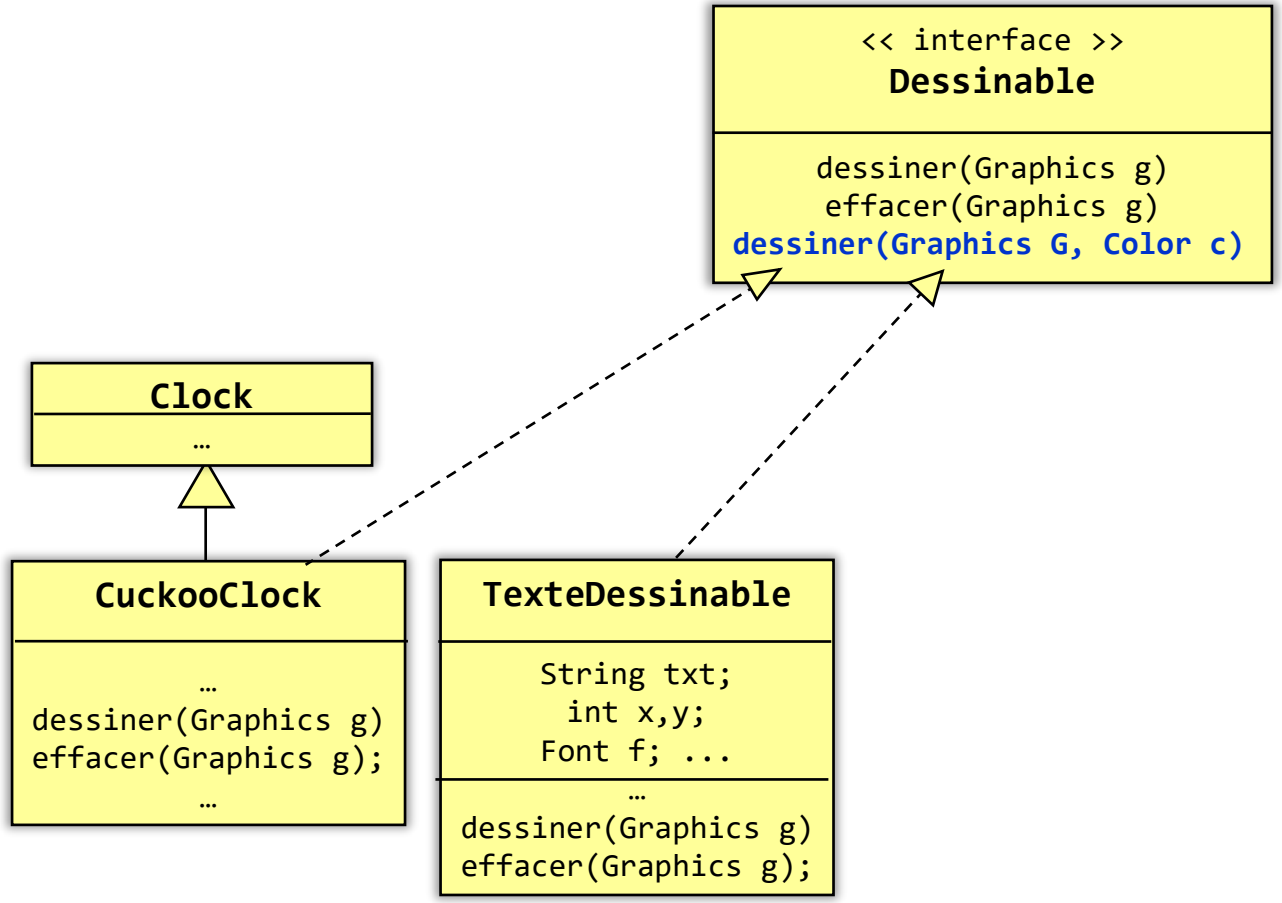



2ème solution :

Définir une sous interface

# Evolution des interfaces

- Quand on définit des interfaces il faut être prudent : tout ajout ultérieur "brise" le code des classes qui implémentent l'interface.




3ème solution :

Définir une méthode par défaut

# Java 8 : quoi de neuf dans les interfaces ? *

- Java7-
  - une méthode déclarée dans une interface ne fournit pas d'implémentation
  - ce n'est qu'une signature, un contrat auquel chaque classe dérivée doit se conformer en fournissant une implémentation propre

- Java 8+ relaxe cette contrainte, possibilité de définir
  - des méthodes statiques
  - des méthodes par défaut
  - des interface fonctionnelles


* titre inspiré du titre de l'article *Java 8 : du neuf dans les interfaces !* du blog d'Olivier Croisier
http://thecodersbreakfast.net/index.php?post/2014/01/20/Java8-du-neuf-dans-les-interfaces

# Interfaces Java 8 : méthodes par défaut

- déclaration d'une méthode par défaut
  - fournir un corps à la méthode
  - qualifier la méthode avec le mot clé `default`

```
public interface Foo {
   public default void foo() {
     System.out.println("Default implementation of foo()");
   }
}
```

- les classes filles sont libérées de fournir une implémentation d'une méthode `default`, en cas d'absence d'implémentation spécifique c'est la méthode par défaut qui est invoquée

```
public interface Itf {

    /** Pas d'implémentation - comme en Java 7
        et antérieur */
    public void foo();

    public default void bar() {
        System.out.println("Itf -> bar() [default]");
    }

    public default void baz() {
        System.out.println("Itf -> baz() [default]");
    }
}
```

```
public class Cls implements Itf {

    @Override
    public void foo() {
        System.out.println("Cls -> foo()");
    }

    @Override
    public void bar() {
        System.out.println("Cls -> bar()");
    }
}
```

```
Cls cls = new Cls();
cls.foo();
cls.bar();
cls.baz();
```

→

```
Cls -> foo()
Cls -> bar()
Itf -> baz() [default]
```

# Interfaces Java 8 : méthodes par défaut

- mais qu'en est-il de l'héritage en diamant ?

```
public interface InterfaceA {
    public default void foo() {
        System.out.println("A -> foo()");
    }
}
```

```
public interface InterfaceB {
    public default void foo() {
        System.out.println("B -> foo()");
    }
}
```

```
public class Cls implements InterfaceA, InterfaceB {
}
```

Erreur de compilation
"class Test inherits unrelated defaults for foo() from typesInterfaceA and InterfaceB"

```
Cls cls = new Cls();
cls.foo();
```

?

?

```
public class Cls implements InterfaceA, InterfaceB {
 public void foo() {
        System.out.println("Test -> foo()");
    }
}
```

Pour résoudre le conflit, une seule solution : implémenter la méthode au niveau de la classe elle-même, car l'implémentation de la classe est toujours prioritaire.

# Interfaces Java 8 : méthodes par défaut

- mais qu'en est-il de l'héritage en diamant ?

```
public interface InterfaceA {
    public default void foo() {
        System.out.println("A -> foo()");
    }
}
```

```
public interface InterfaceB {
    public default void foo() {
        System.out.println("B -> foo()");
    }
}
```

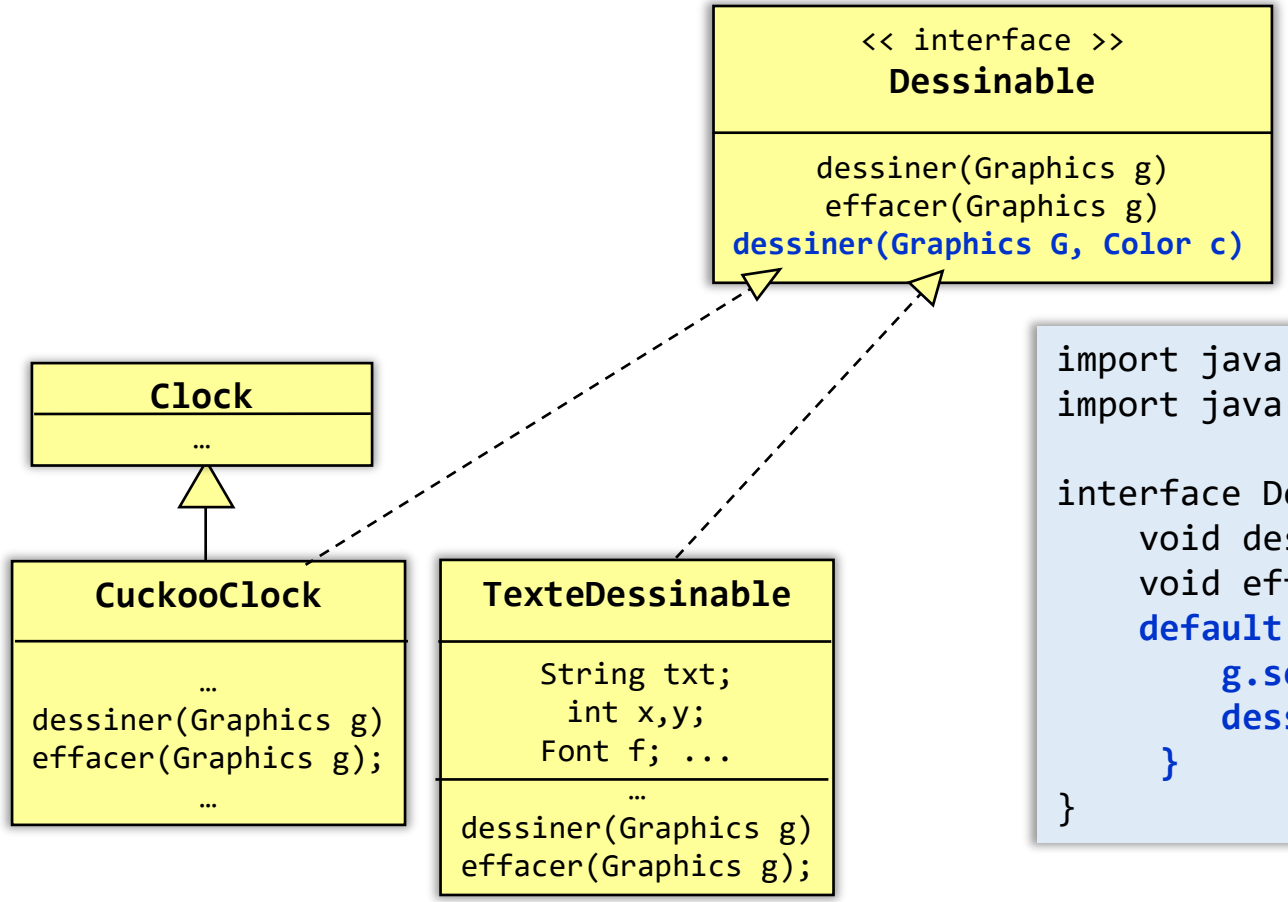
```
public class Cls implements InterfaceA, InterfaceB {
  public void foo() {
      InterfaceB.super.foo();
  }
}
```

```
Cls cls = new Cls();
cls.foo();
```

Possibilité d'accéder sélectivement aux implémentations par défaut :
*nomInterface*.**super**.*méthode*

# Evolution des interfaces

- Quand on définit des interfaces il faut être prudent : tout ajout ultérieur "brise" le code des classes qui implémentent l'interface.




3ème solution :

Définir une méthode par défaut


```
import java.awt.Color;
import java.awt.Graphics;

interface Dessinable {
    void dessiner(Graphics g);
    void effacer(Graphics g);
    default void dessiner(Graphics g, color c) {
        g.setColor(c);
        dessiner(g);
     }
}
```