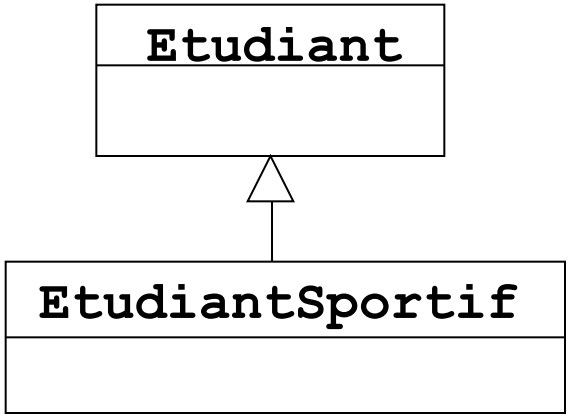
Heritage et Polymorphisme

Surclassement

- La réutilisation du code est un aspect important de l'héritage, mais ce n'est peut être pas le plus important
- Le deuxième point **fondamental** est la relation qui relie une classe à sa super-classe :

Une classe B qui hérite de la classe A peut être vue comme un sous-type (sous ensemble) du type défini par la classe A.



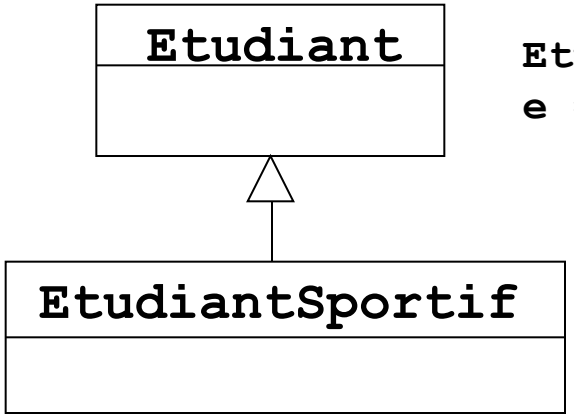
Un `EtudiantSportif` est un `Etudiant`

L'ensemble des étudiants sportifs est inclus dans l'ensemble des étudiants

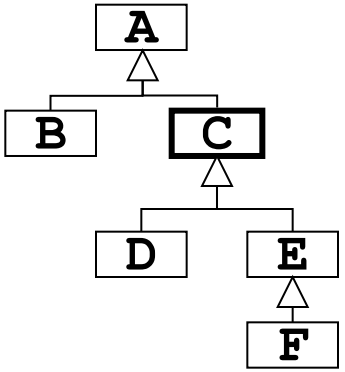
Surclassement

- tout objet instance de la classe **B** peut être aussi vu comme une instance de la classe **A**.

- Cette relation est directement supportée par le langage JAVA :
  - *à une référence déclarée de type A il est possible d'affecter une valeur qui est une référence vers un objet de type B (**surclassement** ou **upcasting**)*




```
Etudiant e;
e = new EtudiantSportif(...);
```




```
C c;
c = new D();
c = new E();
c = new F();
c = new A();
c = new B();
```

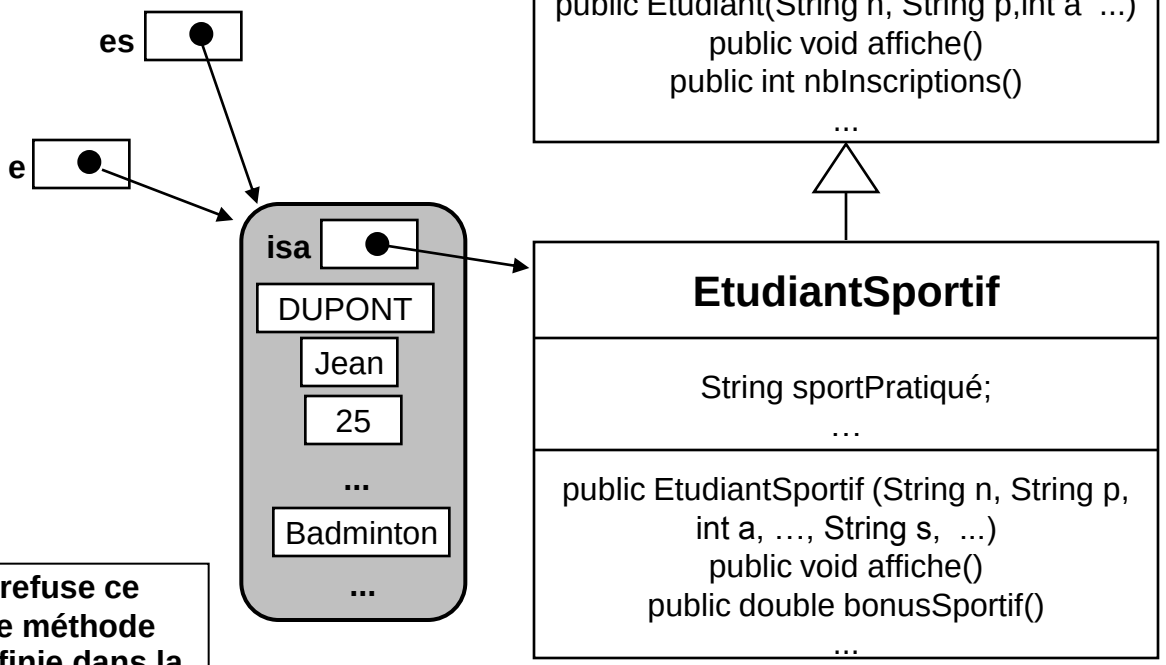
- *plus généralement à une référence d'un type donné, il est possible d'affecter une valeur qui correspond à une référence vers un objet dont le type effectif est n'importe quelle sous-classe directe ou indirecte du type de la référence*

Surclassement

- Lorsqu'un objet est "sur-classé" il est vu par le compilateur comme un objet du type de la référence utilisée pour le désigner
  - *Ses fonctionnalités sont alors restreintes à celles proposées par la classe du type de la référence*

```
EtudiantSportif es;
es = new EtudiantSportif("DUPONT","Jean",
                         25,..,"Badminton",..);

 Etudiant e;
 e = es;  // upcasting

 e.affiche();
 es.affiche();

 e.nbInscriptions();
 es.nbInscriptions();

 es.bonusSportif();
 e.bonusSportif();
```

Le compilateur refuse ce message : pas de méthode `bonusSportif` définie dans la classe Etudiant



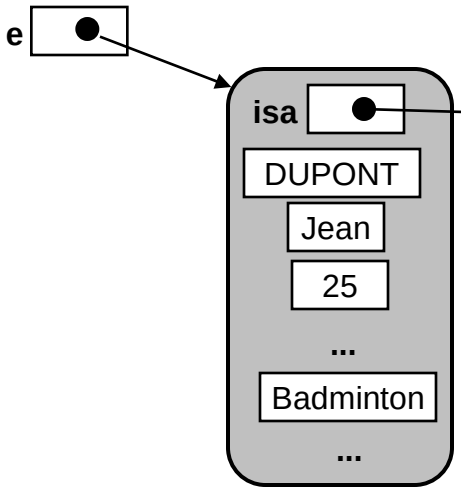
Lien dynamique

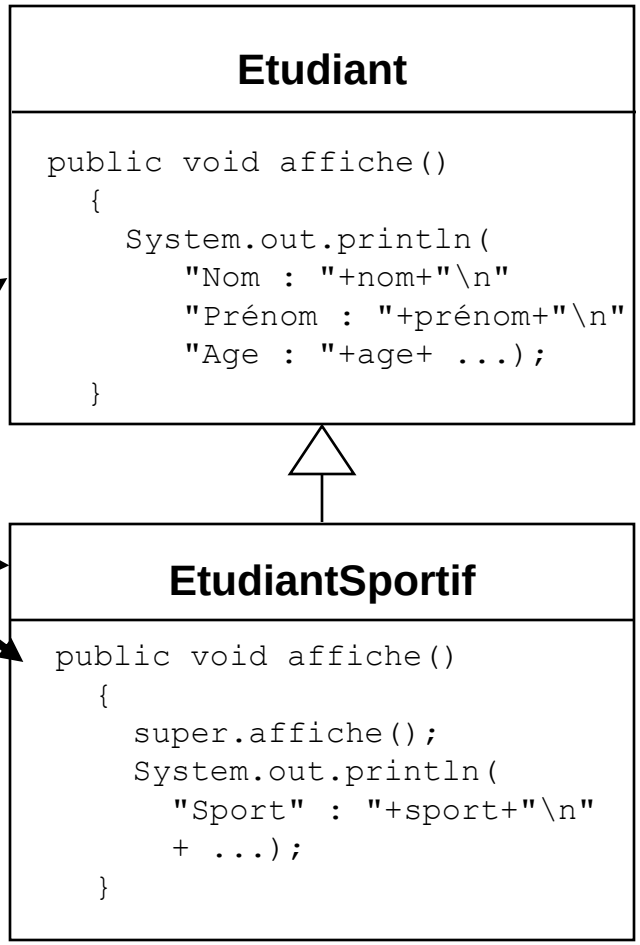
Résolution des messages

- Que va donner **`e.affiche()`** ?

```
Etudiant e = new EtudiantSportif(
     "DUPONT","Jean",25,..,
     "Badminton",..);
```

```
e.affiche();
```





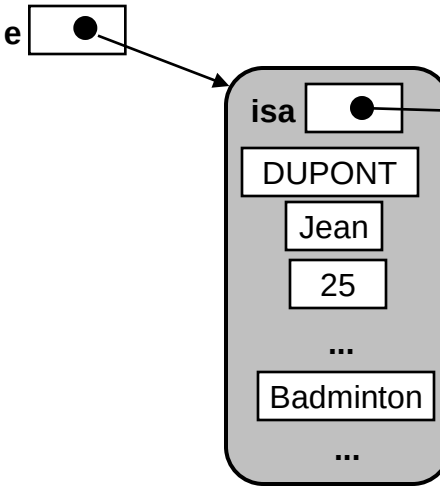
Lien dynamique

Résolution des messages

Lorsqu'une méthode d'un objet est accédée au travers d'une référence "surclassée", c'est la méthode telle qu'elle est définie au niveau de la classe effective de l'objet qui est en fait invoquée et exécutée

```
Etudiant e = new EtudiantSportif(
     "DUPONT","Jean",25,..,
     "Badminton",..);
```

```
e.affiche();
```




```
Nom : DUPONT
Prénom : Jean
Age : 25
...
Sport : Badminton
```

Etudiant

```
public void affiche()
  {
    System.out.println(
       "Nom : "+nom+"\n"
       "Prénom : "+prénom+"\n"
       "Age : "+age+ ...);
  }
```

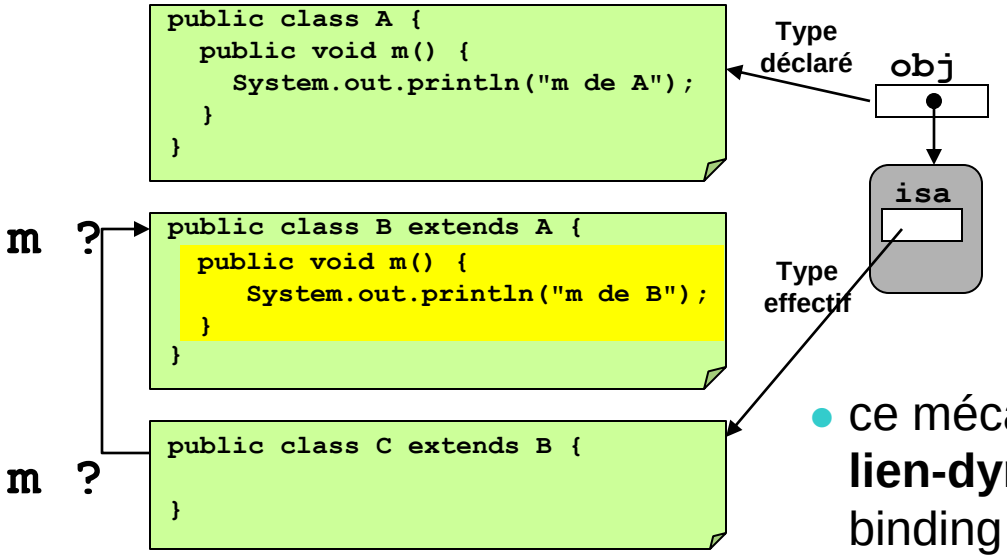
EtudiantSportif

```
public void affiche()
  {
    super.affiche();
    System.out.println(
      "Sport" : "+sport+"\n"
      + ...);
  }
```

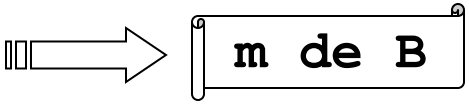
Lien dynamique

Mécanisme de résolution des messages

- Les messages sont résolus **à l'exécution**
  - *la méthode exécutée est déterminée à l'exécution (run-time) et non pas à la compilation*
  - *à cet instant le type exact de l'objet qui reçoit le message est connu*
    - *la méthode définie pour le type réel de l'objet recevant le message est appelée (et non pas celle définie pour son type déclaré).*




```
A obj = new C();
obj.m();
```



- ce mécanisme est désigné sous le terme de **lien-dynamique** (dynamic binding, late-binding ou run-time binding)

Lien dynamique

Vérifications statiques

- **A la compilation**: seules des **vérifications statiques** qui se basent sur le type déclaré de l'objet (de la référence) sont effectuées
  - *la classe déclarée de l'objet recevant un message doit posséder une méthode dont la signature correspond à la méthode appelée.*

```
public class A {
  public void m1() {
    System.out.println("m1 de A");
  }
}
```

```
public class B extends A {
  public void m1() {
    System.out.println("m1 de B");
  }
  public void m2() {
    System.out.println("m2 de B");
  }
}
```

Type déclaré **obj**

```
A obj = new B();
obj.m1();
obj.m2();
```

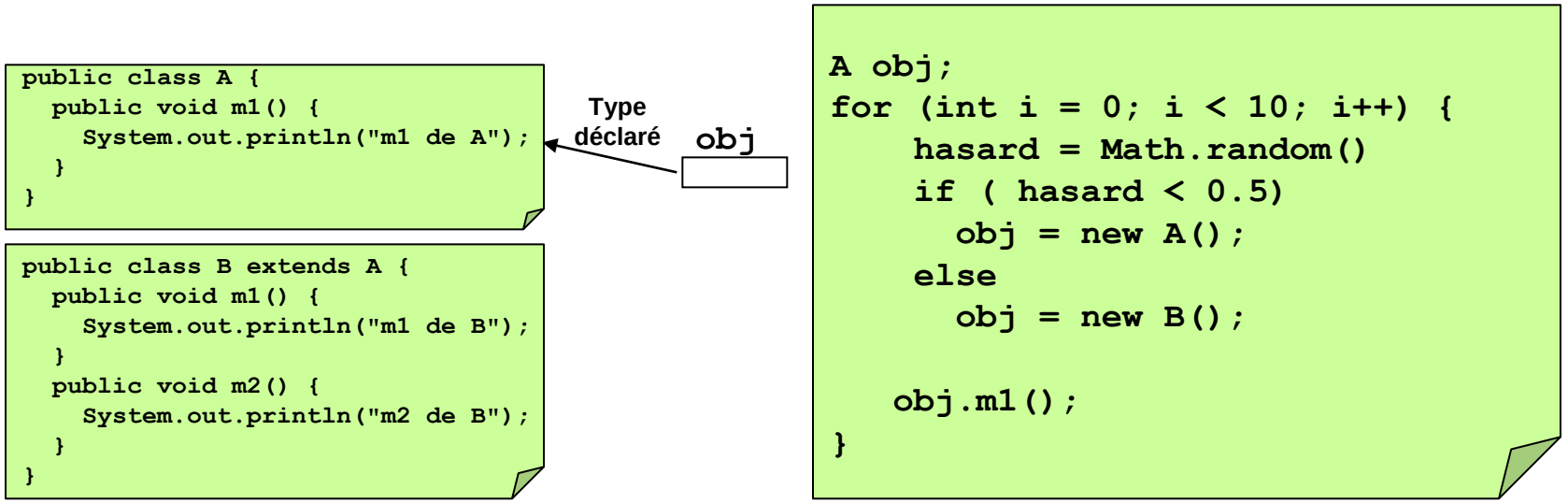
```
Test.java:21: cannot resolve symbol
symbol  : method m2  ()
location: class A
    obj.m2();
       ^
1 error
```

- garantir dès la compilation que les messages pourront être résolus au moment de l'exécution → robustesse du code

Lien dynamique

Vérifications statiques

- à la compilation il n'est pas possible de déterminer le type exact de l'objet récepteur d'un message



- vérification statique: garantit dès la compilation que les messages pourront être résolus au moment de l'exécution

Lien dynamique

Choix des methodes, sélection du code

Le choix de la méthode à exécuter est effectué **statiquement à la compilation** en fonction du type des paramètres

```
public class A {
  public void m1() {
    System.out.println("m1 de A");
  }

  public void m1(int x) {
    System.out.println("m1(x) de A");
  }
}
```

```
public class B extends A {
  public void m1() {
    System.out.println("m1 de B");
  }
  public void m2() {
    System.out.println("m2 de B");
  }
}
```

```
invokevirtual … <Method m1()>

invokevirtual … <Method m1(int)>

invokevirtual … <Method m1()>
```

Byte-code

Compilation

```
A refA = new A();

refaA.m1();

refA.m1(10);

refA = new B();

refA.m1();
```

La sélection du code à exécuter est effectué **dynamiquement à l'exécution** en fonction du type effectif du récepteur du message

Polymorphisme

A quoi servent l'upcasting et le lien dynamique ?

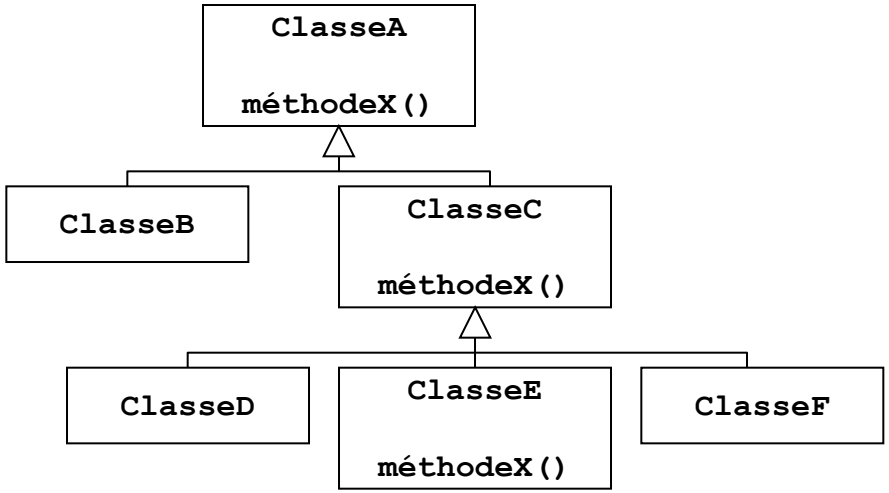
A la mise en œuvre du polymorphisme

- Le terme polymorphisme décrit la caractéristique d'un élément qui peut se présenter sous différentes formes.

- En programmation Objet, on appelle polymorphisme
  - *le fait qu'un objet d'une classe puisse être manipulé comme s'il appartenait à une autre classe.*
  - *le fait que la même opération puisse se comporter différemment sur différentes classes de la hiérarchie.*

- **"**Le **polymorphisme** constitue la troisième caractéristique essentielle d'un langage orienté objet après l'abstraction des données (encapsulation) et l'héritage"
  Bruce Eckel "Thinking in JAVA"

Polymorphisme




```
ClasseA objA;
objA = ...
objA.methodeX();
```

Surclassement
la référence peut désigner des objets de classe différente (n'importe quelle sous classe de ClasseA)

+

Lien dynamique
Le comportement est différent selon la classe effective de l'objet

un cas particulier de polymorphisme

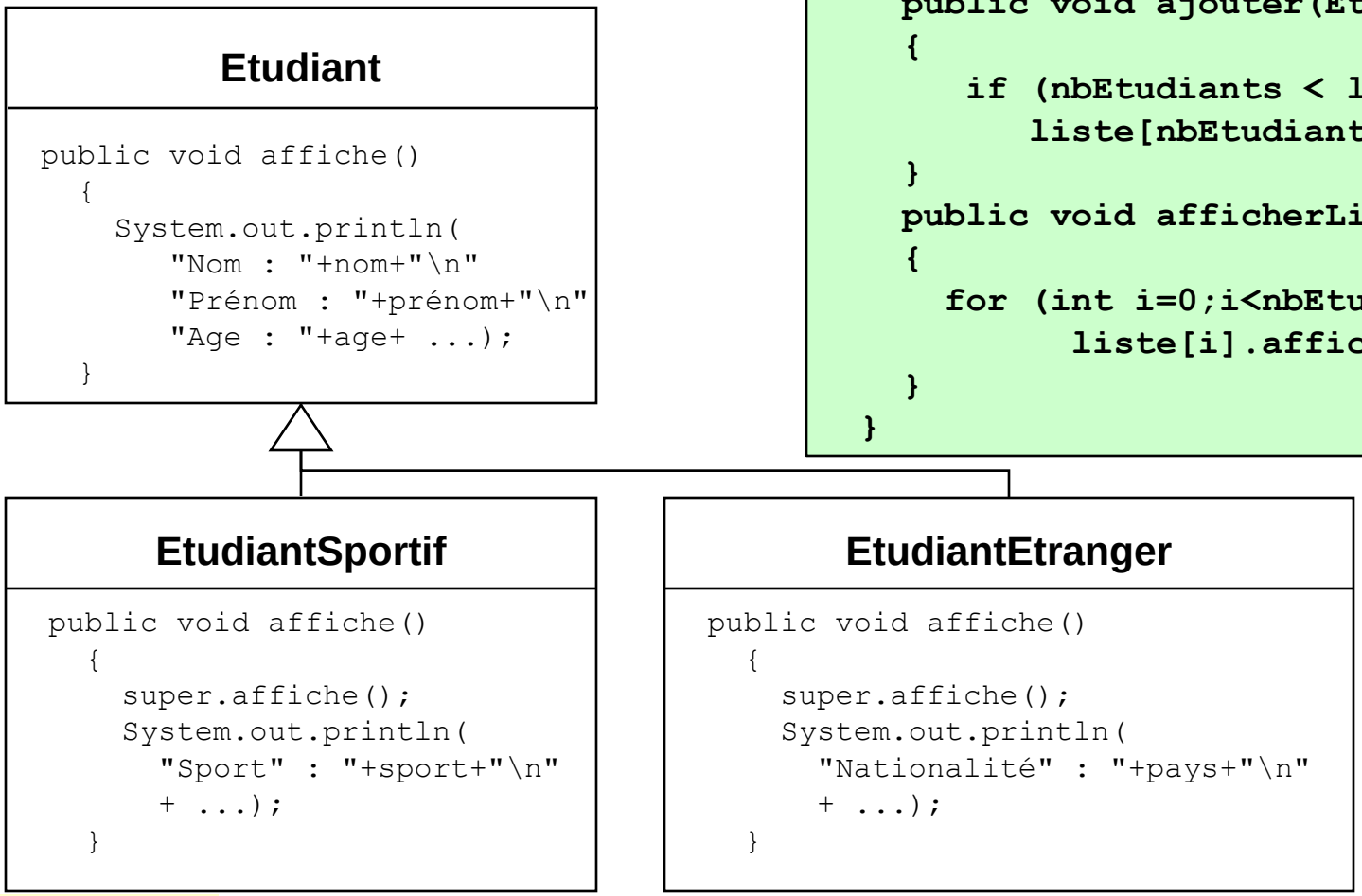
manipulation uniforme des objets de plusieurs classes par l'intermédiaire d'une classe de base commune

Polymorphisme

`liste` peut contenir des étudiants de n'importe quel type

```
GroupeTD td1 = new GroupeTD();
td1.ajouter(new Etudiant("DUPONT", …));
td1.ajouter(new EtudiantSportif("BIDULE",
                "Louis", … , "ski alpin");
```

```
public class GroupeTD{
   Etudiant[] liste = new Etudiant[30];
   int nbEtudiants = 0;
   ...
   public void ajouter(Etudiant e)
   {
      if (nbEtudiants < liste.lenght)
         liste[nbEtudiants++] = e;
   }
   public void afficherListe()
   {
     for (int i=0;i<nbEtudiants; i++)
           liste[i].affiche();
   }
 }
```



Si un nouveau type d'étudiant est défini, le code de `GroupeTD` reste inchangé

Polymorphisme

- En utilisant le polymorphisme en association à la liaison dynamique
  - *plus besoin de distinguer différents cas en fonction de la classe des objets*
  - *possible de définir de nouvelles fonctionnalités en héritant de nouveaux types de données à partir d'une classe de base commune sans avoir besoin de modifier le code qui manipule l'interface de la classe de base*

- Développement **plus rapide**
- Plus grande **simplicité** et **meilleure organisation** du code
- Programmes plus facilement **extensibles**
- Maintenance du code **plus aisée**

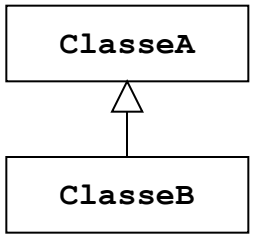
Polymorphisme

Pour conclure

« Once you know that all method binding in Java
happens polymorphically via late binding,
you can always write your code to talk to the base class,
and know that all the derived-class cases will work
correctly using the same code.
Or put it another way, you send a message to an object and let
the object figure out the right thing to do »

Bruce Eckel, Thinking in Java

Surcharge et polymorphisme




```
public class ClasseC {

   public static void methodeX(ClasseA a){
      System.out.println("param typeA");
   }

   public static void methodeX(ClasseB b){
      System.out.println("param typeB");
   }

}
```

surcharge

```
ClasseA refA = new ClasseA();
ClasseC.methodX(refA);
ClasseB refB = new ClasseB();
ClasseC.methodX(refB);
refA = refB; // upCasting
ClasseC.methodX(refA);
```

param TypeA

param TypeB

param TypeA

```
invokestatic … <Method void methodX( ClasseA )>
```

Byte-code

Le choix de la méthode à exécuter est effectué à la compilation en fonction des types déclarés : Sélection statique

A propos de equals

- Tester l'égalité de deux objets de la même classe

```
public class Object {
  ...
 public boolean equals(Object o)
    return this == o;
  }
  ...
}
```

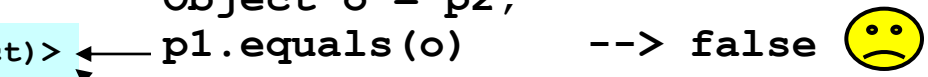
```
public class Point {

      private double x;
      private double y;

      ...

}
```

De manière générale, il vaut mieux éviter de surcharger des méthodes en spécialisant les arguments

```
public boolean equals(Point pt) {
  return this.x == pt.x && this.y == pt.y;
}
```

surcharge (overloads) la méthode equals(Object o) héritée de Object

```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);
p1.equals(p2);   --> true
Object o = p2;
p1.equals(o)     --> false
o.equals(p1)     --> false
```



`invokevirtual … <Method equals(Object)>`

Le choix de la méthode à exécuter est effectué statiquement à la compilation

en fonction du type déclaré de l'objet récepteur du message et du type déclaré du (des) paramètre(s)

A propos de equals

- Tester l'égalité de deux objets de la même classe

```
public class Object {
  ...
 public boolean equals(Object o)
   return this == o
 }
 ...
}
```

```
public class Point {

     private double x;
     private double y;

     ...

}
```

```
@Override
public boolean equals(Object o) {
   if (this == o)
        return true;
   if (! (o instanceof Point))
        return false;
   Point pt = (Point) o; // downcasting
   return this.x == pt.x && this.y == pt.y;
}
```

redéfinir (overrides) la méthode equals(Object o) héritée de Object

```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);
p1.equals(p2)        --> true
Object o = p2;
p1.equals(o)         --> true
o.equals(p1)         --> true
```

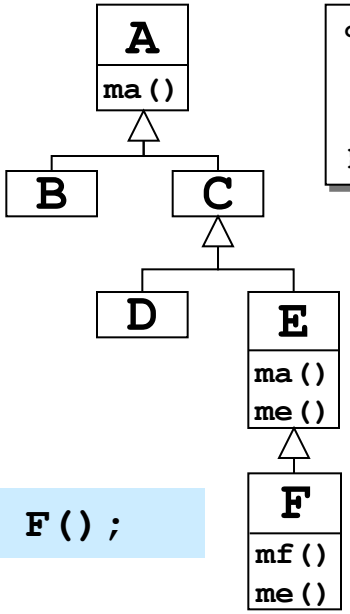
Downcasting

```
ClasseX obj = …
  ClasseA  a = (ClasseA) obj;
```

- Le downcasting (ou transtypage) permet de « forcer un type » à la compilation
  - *C'est une « promesse » que l'on fait au moment de la compilation.*
- Pour que le transtypage soit valide, il faut qu'à l'exécution le type effectif de ***`obj`*** soit « compatible » avec le type ***`ClasseA`***
  - *Compatible : la même classe ou n'importe quelle sous classe de* ***`ClasseA`*** *(`obj instanceof ClasseA`)*
- Si la promesse n'est pas tenue une erreur d'exécution se produit.
  - *ClassCastException est levée et arrêt de l'exécution*

```
java.lang.ClassCastException: ClasseX
        at Test.main(Test.java:52)
```

Upcasting/Downcasting




```
class A {
    public void ma() {
        System.out.println("methode ma définie dans A");
    }
}
```

```
class E extends C {
    public void ma() {
        System.out.println("methode ma redéfinie dans E");
    }
    public void me() {
        System.out.println("methode me définie dans E");
    }
}
```

```
class F extends E {
    public void mf() {
        System.out.println("methode mf définie dans f");
    }
    public void me() {
        System.out.println("methode me redéfinie dans F");
    }
}
```

```
C c = new F();
```

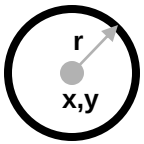
	compilation	exécution
`c.ma();`	☺ La classe C hérite d'une méthode ma	☺ → méthode ma définie dans E
`c.mf();`	☹ *Cannot find symbol : metod mf()* Pas de méthode mf() définie au niveau de la classe C	
`B b = c;`	☹ *Incompatible types* Un C n'est pas un B	
`E e = c;`	☹ *Incompatible types* Un C n'est pas forcément un E	
`E e = (E)c;` `e.me();`	☺ Transtypage (Dowcasting), le compilateur ne fait pas de vérification La classe E définit bien une méthode me	☺ → méthode me définie dans F
`D d = (D) c;`	☺ Transtypage (Dowcasting), le compilateur ne fait pas de vérification	☹ *ClassCastException* Un F n'est pas un D

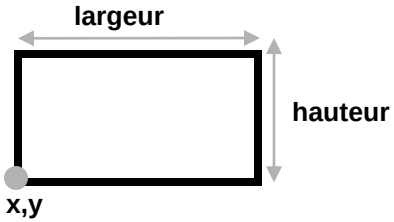
Héritage et abstraction
classes abstraites

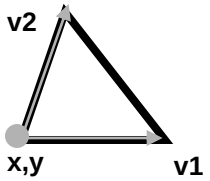
Classes Abstraites

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; //1 des sommets
double x1,y1; // v1
double x2,y2; // v2
Méthodes :
deplacer(double dx, double dy)
double surface()
double périmètre();

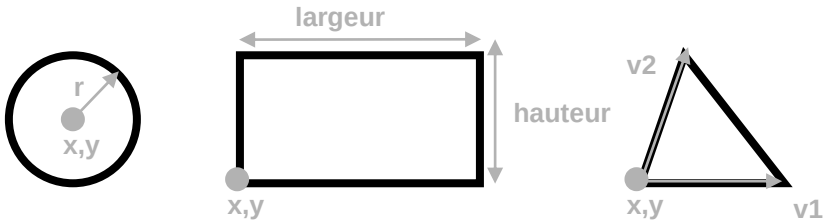
Factoriser le code ?

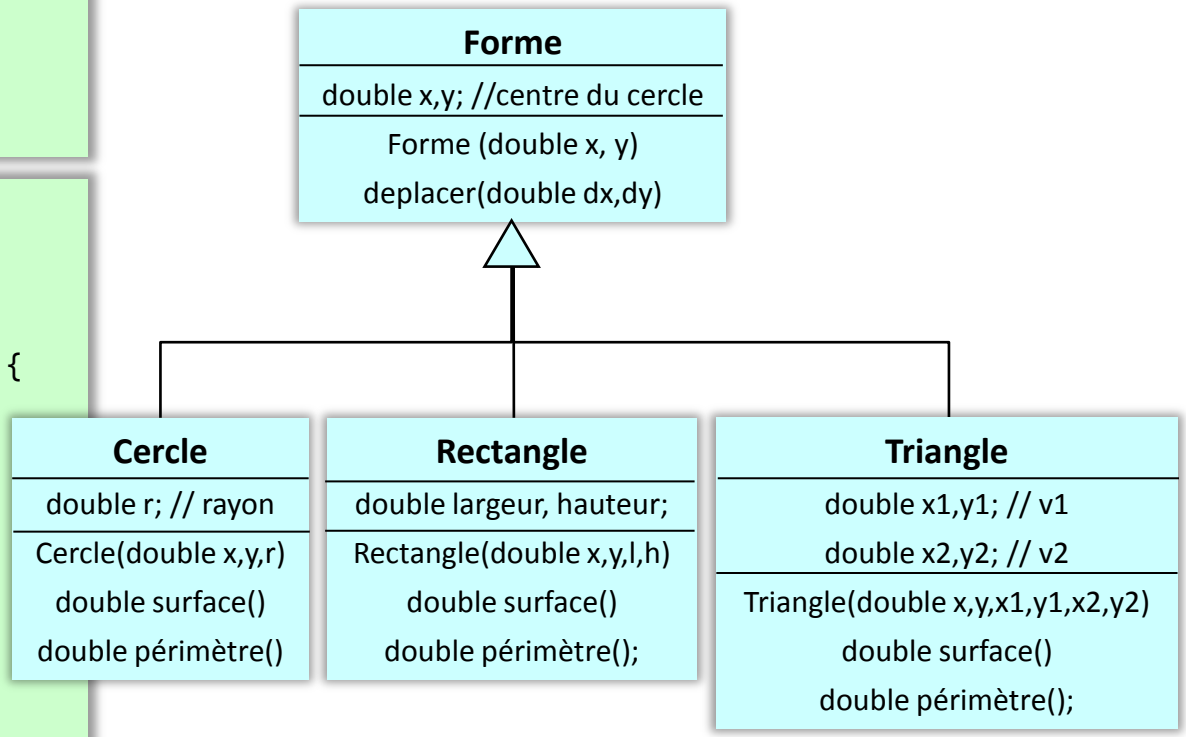
Classes abstraites

```
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;
    }
}
```

Exemple introductif





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].deplace(dx,dy);
  }


  public double périmetreTotal() {
    double pt = 0.0;
    for (int i=0; i < nbFormes++; i++)
       pt += tabFormes[i].périmetre();
    return pt;
  }

}
```

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

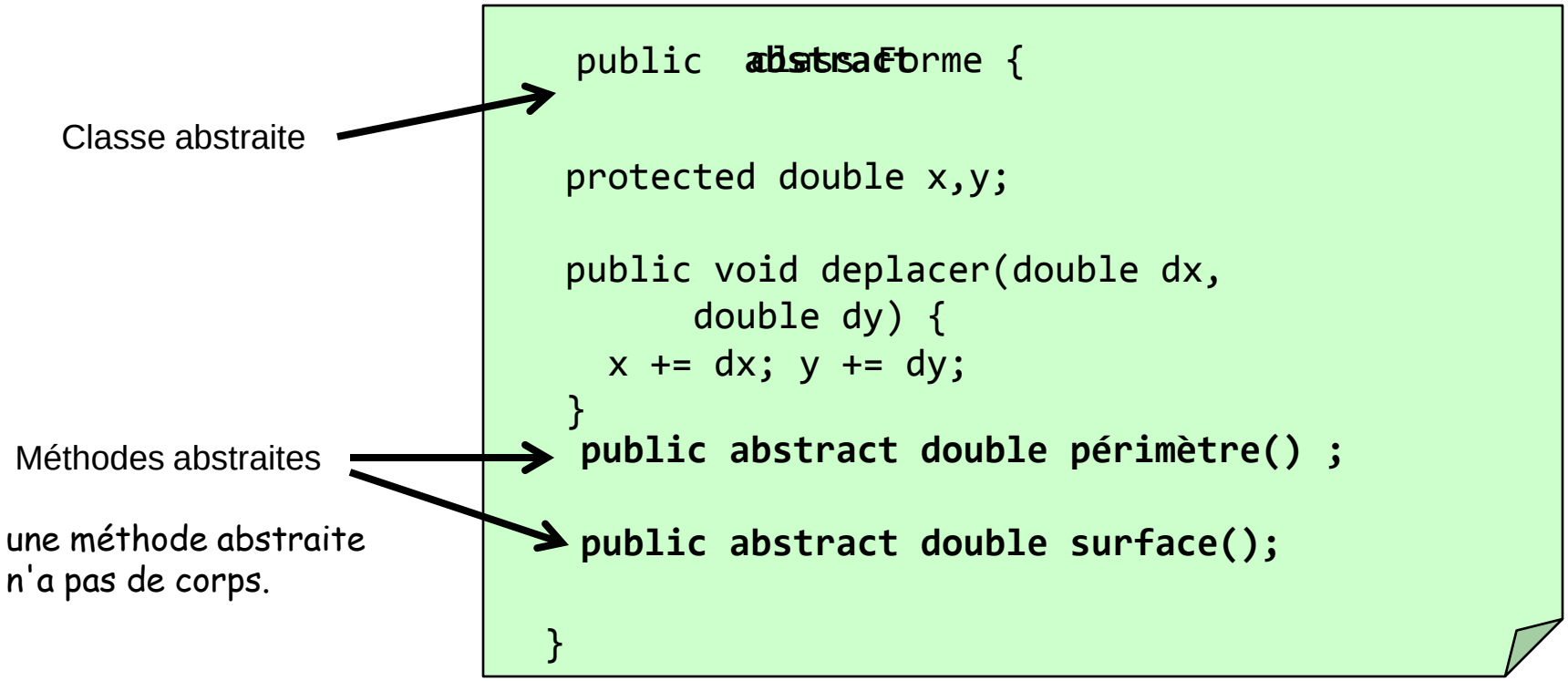
Définir une méthode `périmètre` dans `Forme` ?

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

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

Classes abstraites

- **Utilité** :
  - *définir des concepts incomplets qui devront être implémentés dans les sous classes*
  - *factoriser le code*



Classes abstraites

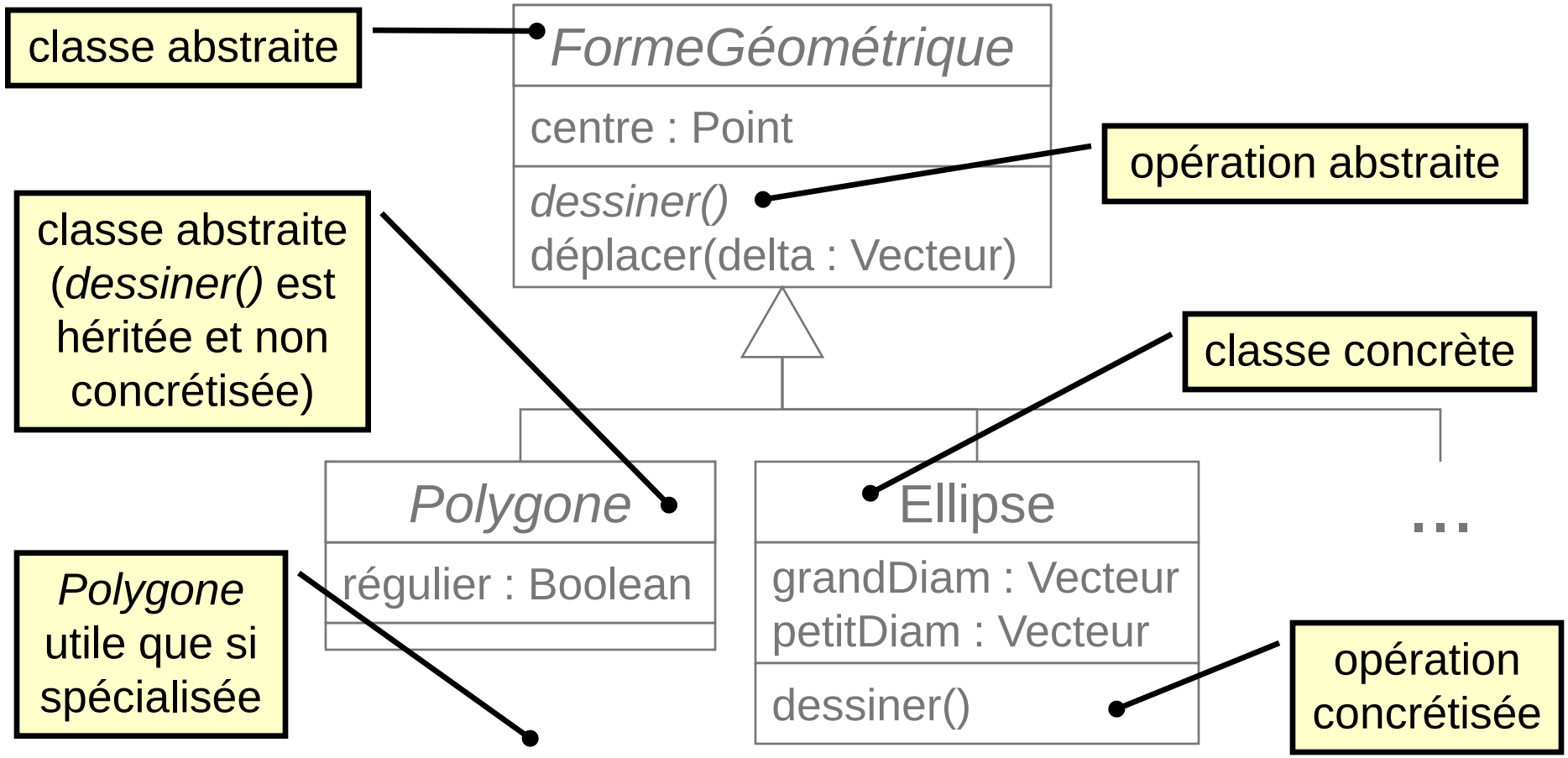
- ***classe abstraite*** : classe non instanciable, c'est à dire qu'elle n'admet pas d'instances directes.
  - *Impossible de faire* `new ClasseAbstraite(…);`
- ***opération abstraite*** : opération n'admettant pas d'implémentation
  - *au niveau de la classe dans laquelle elle est déclarée, on ne peut pas dire comment la réaliser.*
- Une classe pour laquelle au moins une opération abstraite est déclarée est une classe abstraite (l'inverse n'est pas vrai).
- 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

- 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.
- 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 le plus souvent l'utilisation de classes abstraites.

Classes abstraites

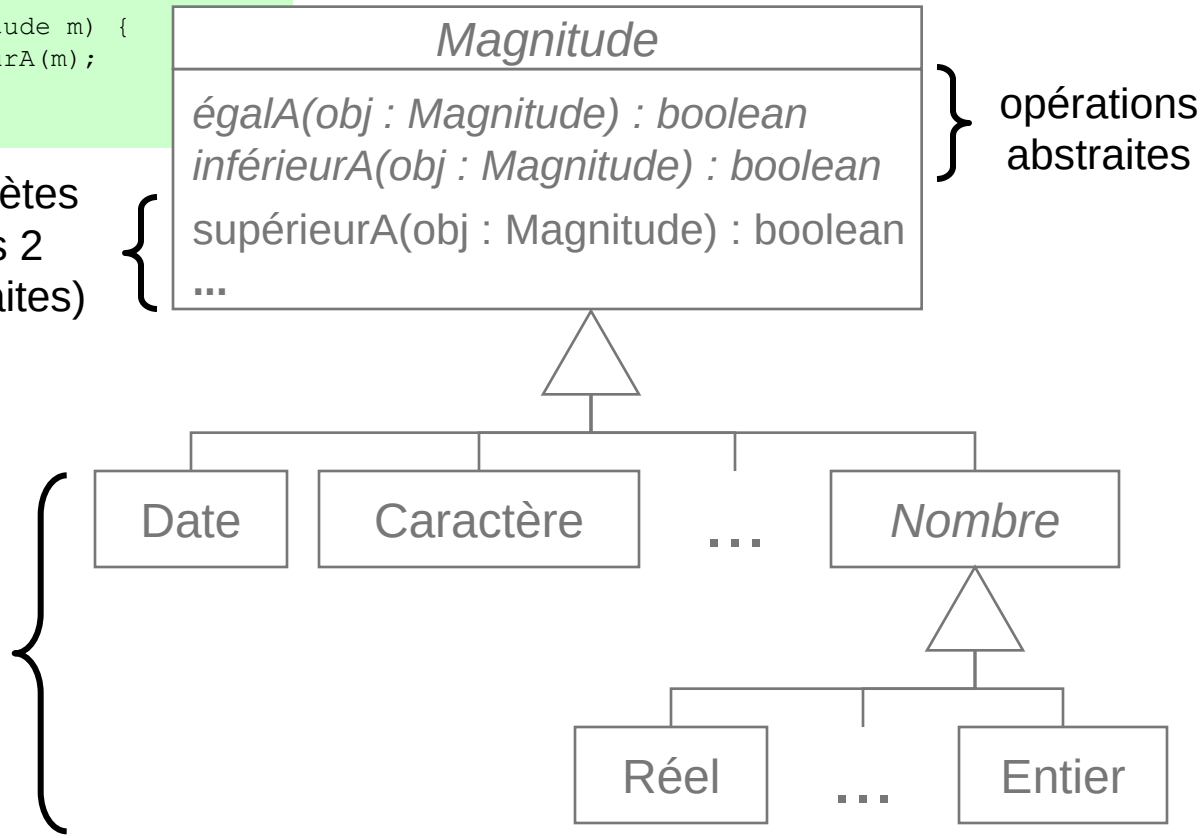
Mieux structurer avec des classes et des opérations abstraites



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

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

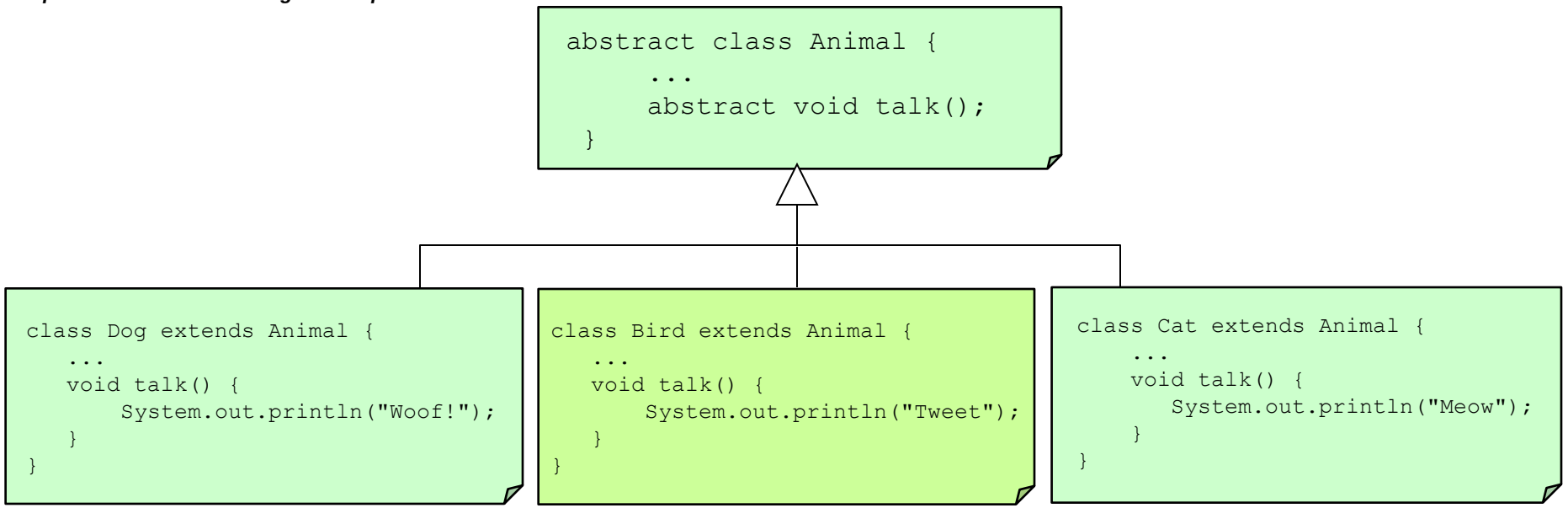
Interfaces

Exemple introductif

Bill Venners ***Designing with Interfaces***

One Programmer's Struggle to Understand the Interfaces

http://www.atrima.com/designtechniques/index.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

```
Animal animal = new Dog();
...
animal = new Cat();
```

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

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

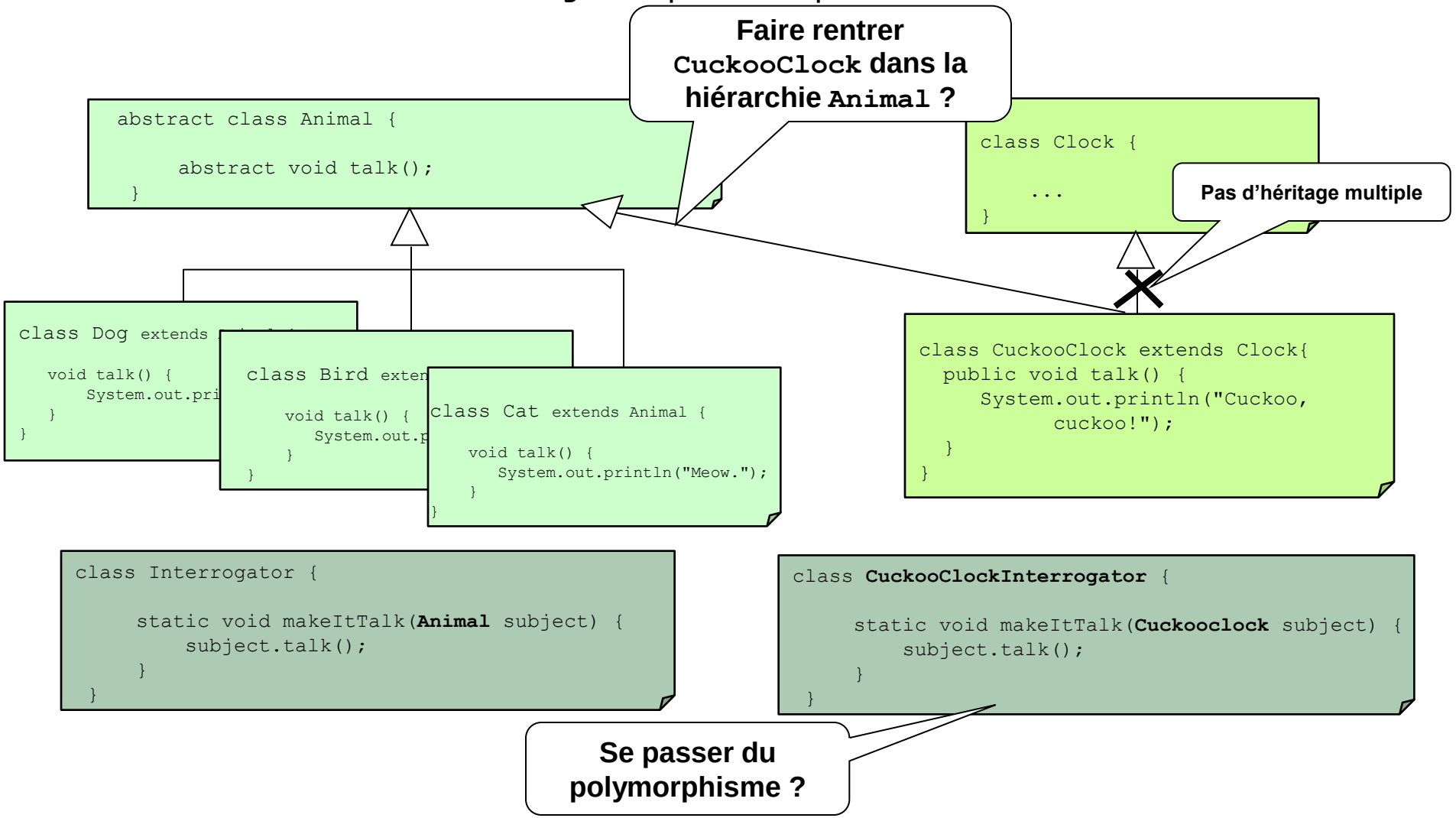
```
class Interrogator {

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

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

Interfaces

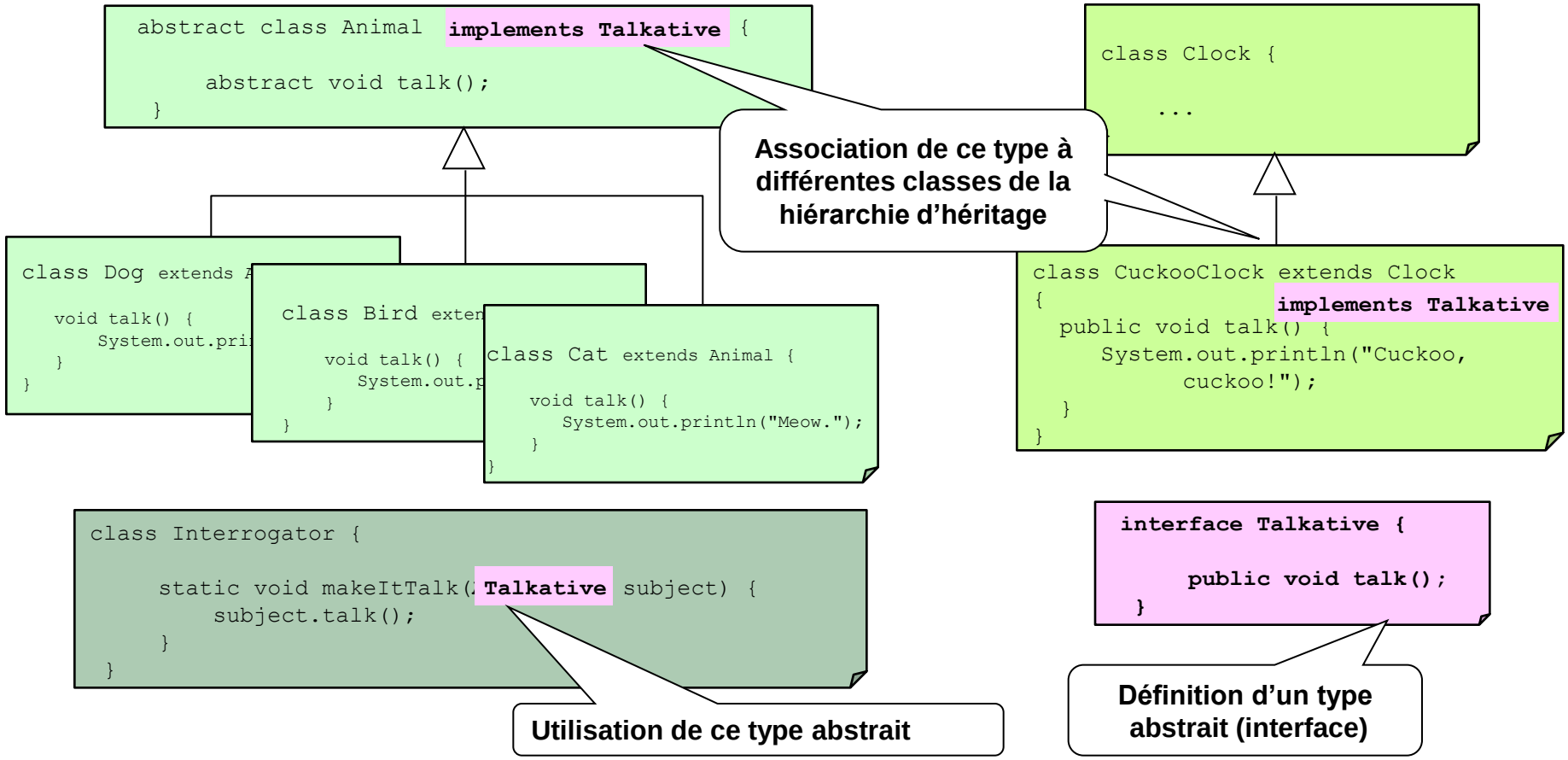
Exemple introductif

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



Interfaces

Exemple introductif



- 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 100% classe abstraite sans attributs et dont toutes les opérations sont abstraites.

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

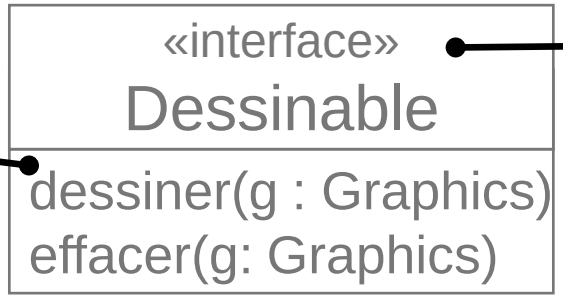
```
package m2pcci.dessin;
import java.awt.Graphics;

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

• Toutes les méthodes sont abstraites

• Elles sont implicitement publiques

Dessinable.java

•Une interface publique doit être définie dans un fichier .java de même nom



interface

opérations abstraites

Interfaces

déclaration d'une interface

- Possibilité de définir des attributs à condition qu'il s'agisse d'attributs de type primitif
- Ces attributs sont implicitement déclarés comme **static final**

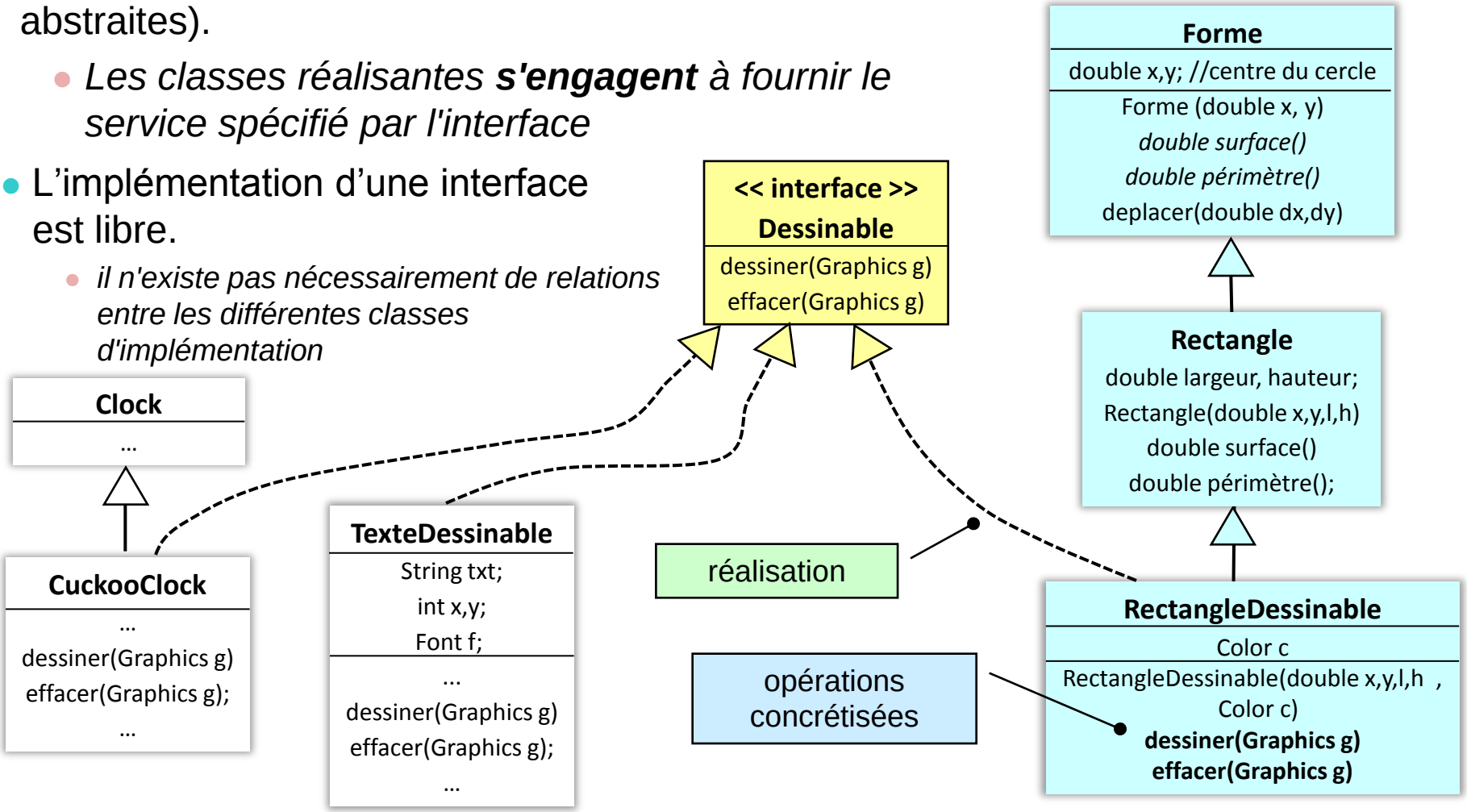
```
import java.awt.Graphics;
public interface Dessinable {

  public static final int MAX_WIDTH = 1024;
  int MAX_HEIGHT = 768;

   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*

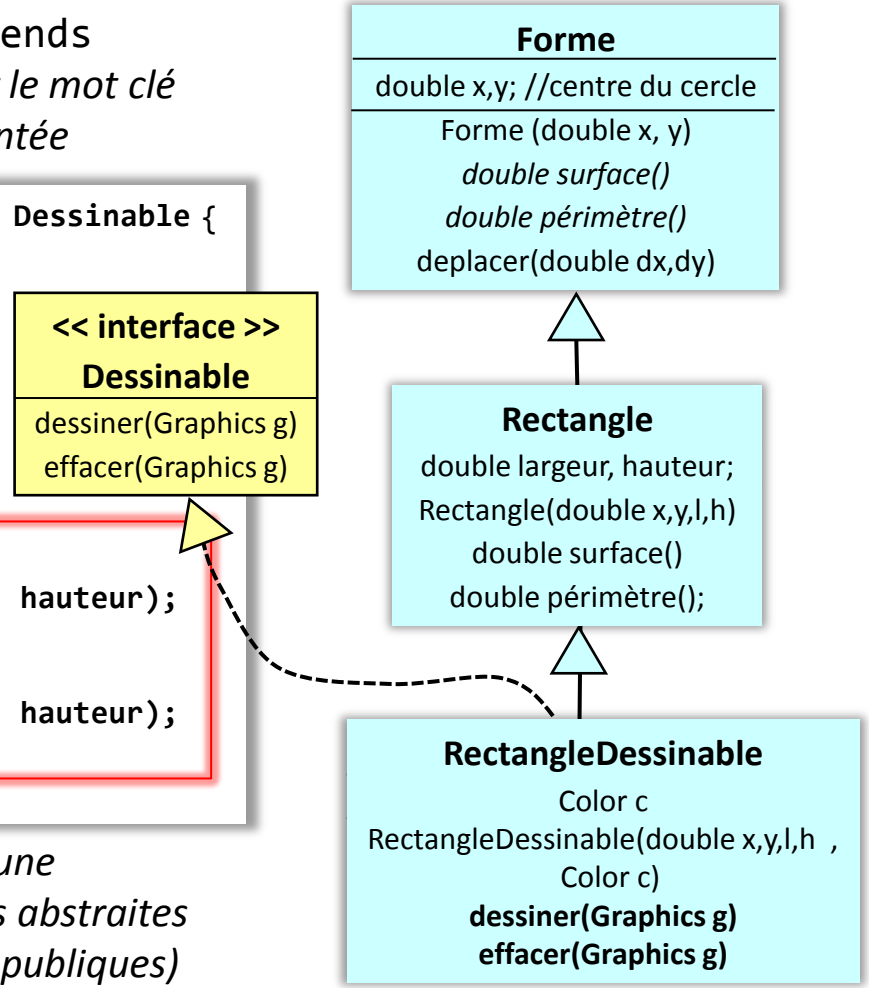


Interfaces

"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);
    }
}
```



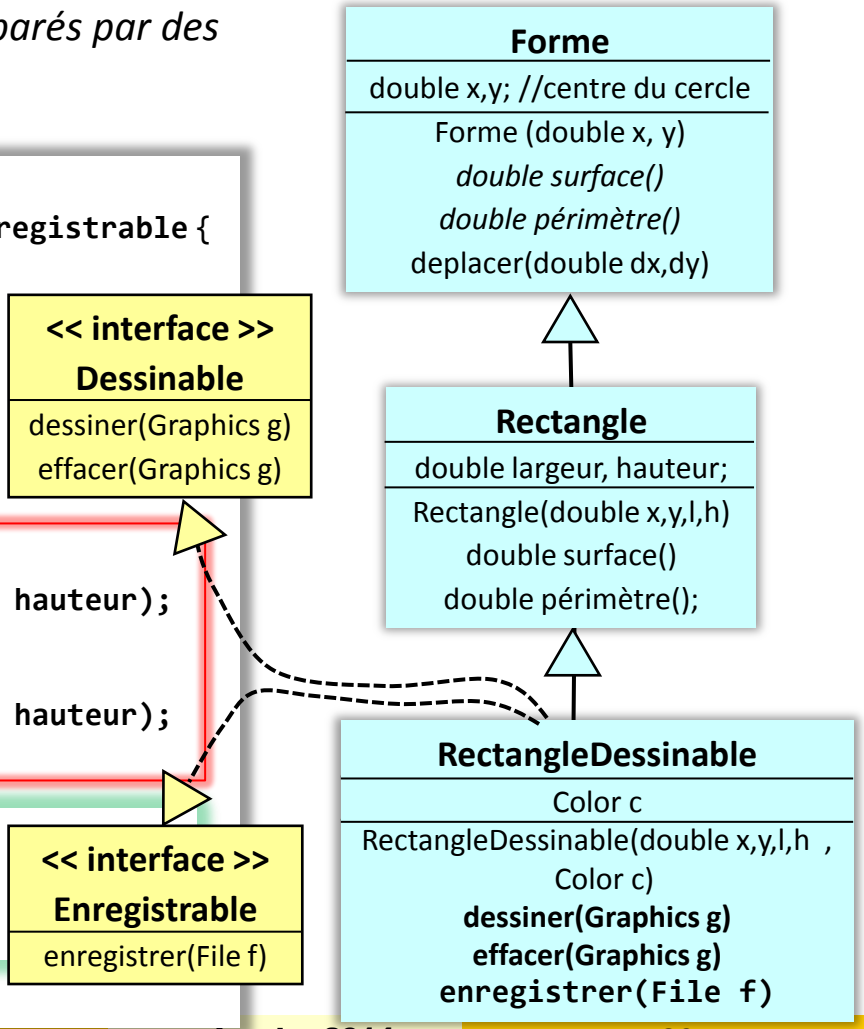
  - *si la classe est une classe concrète* ***elle doit*** *fournir une implémentation (un corps) à chacune des méthodes abstraites définies dans l'interface (qui doivent être déclarées publiques)*

Interfaces

"réalisation" d'une interface

- 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) {
       ...
    }
}
```

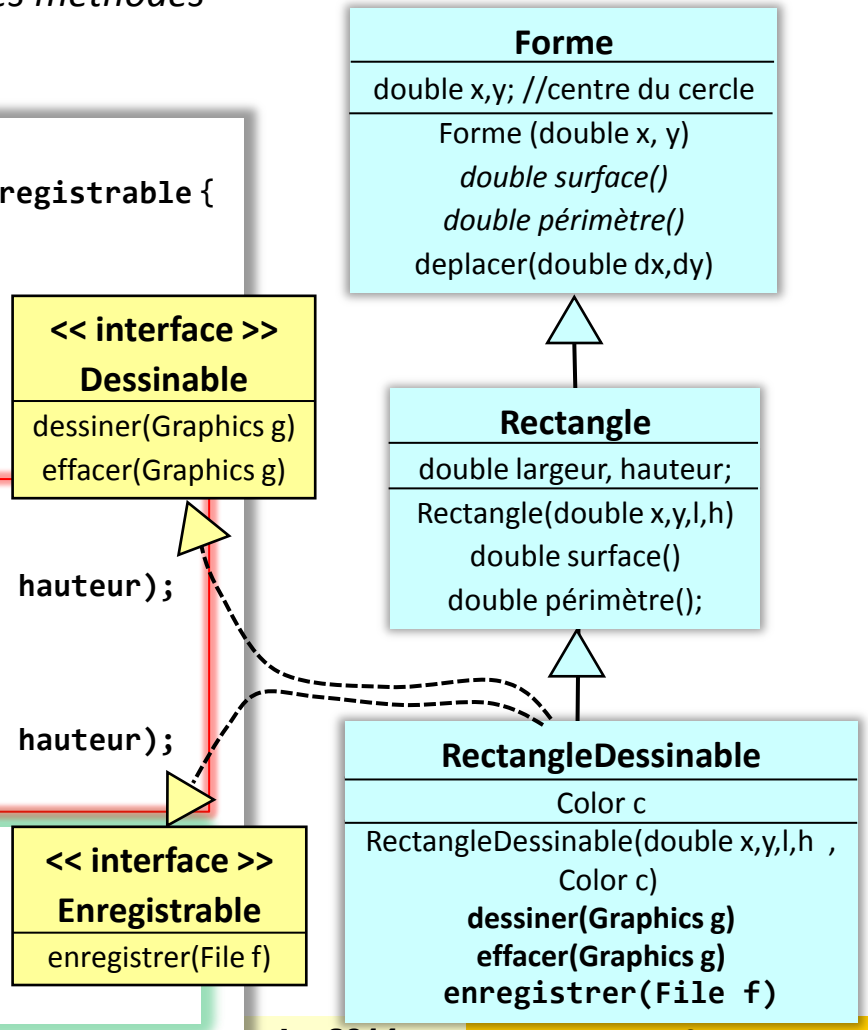


Interfaces

"réalisation" d'une interface

- *pour éviter des redéfinitions 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) {
       ...
    }

}
```



Interfaces

Interface et polymorphisme

- **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();
```

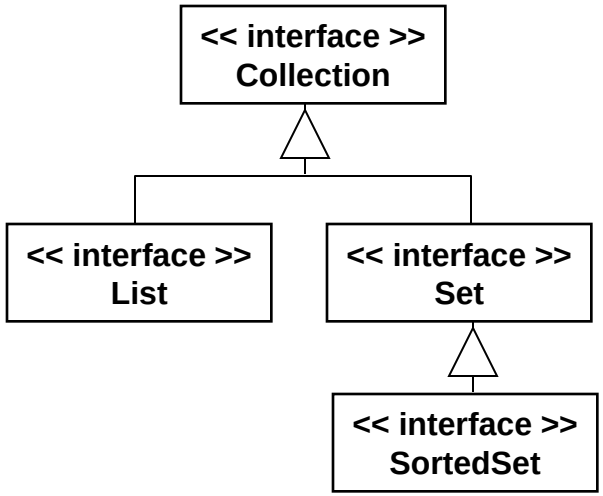
permet de s'intéresser uniquement à certaines caractéristiques d'un objet

règles du polymorphisme s'appliquent de la même manière que pour les classes :
• vérification statique du code
• liaison dynamique

Interfaces

héritage d'interface

- 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*




```
interface Set extends Collection
{
    ...

}
```

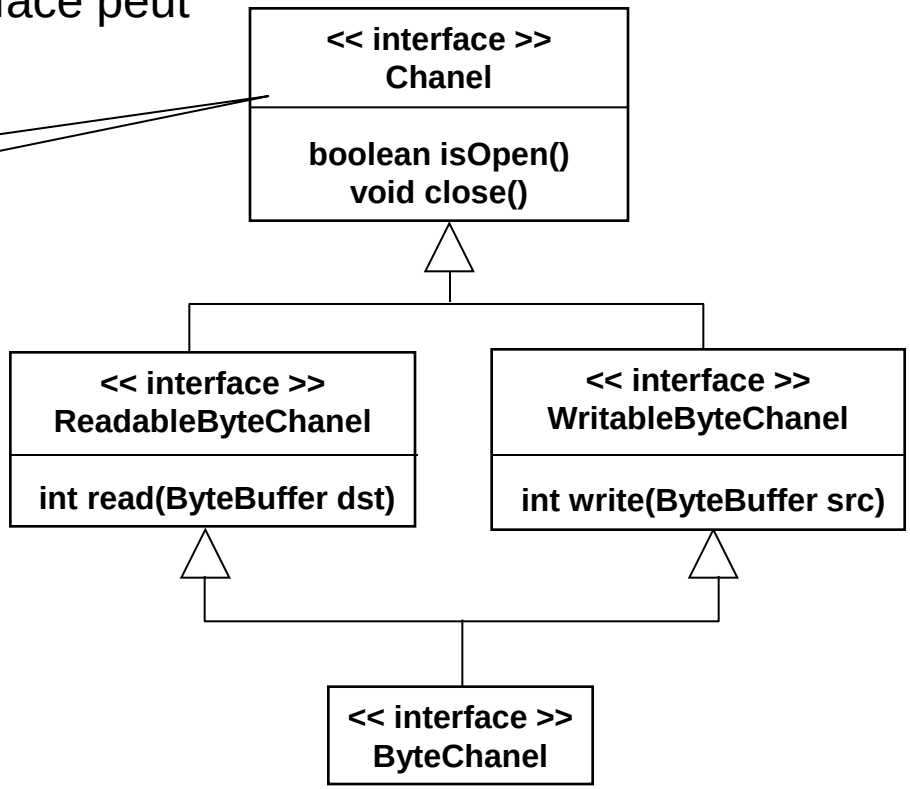
- Une classe qui implémente une interface doit implémenter toutes les méthodes abstraites définies dans l'interface et dans les interfaces dont elle hérite.

Interfaces

héritage d'interfaces

- A la différence des classes une interface peut étendre plus d'une interface à la fois

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




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

Interfaces

interêt

- 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

Interfaces

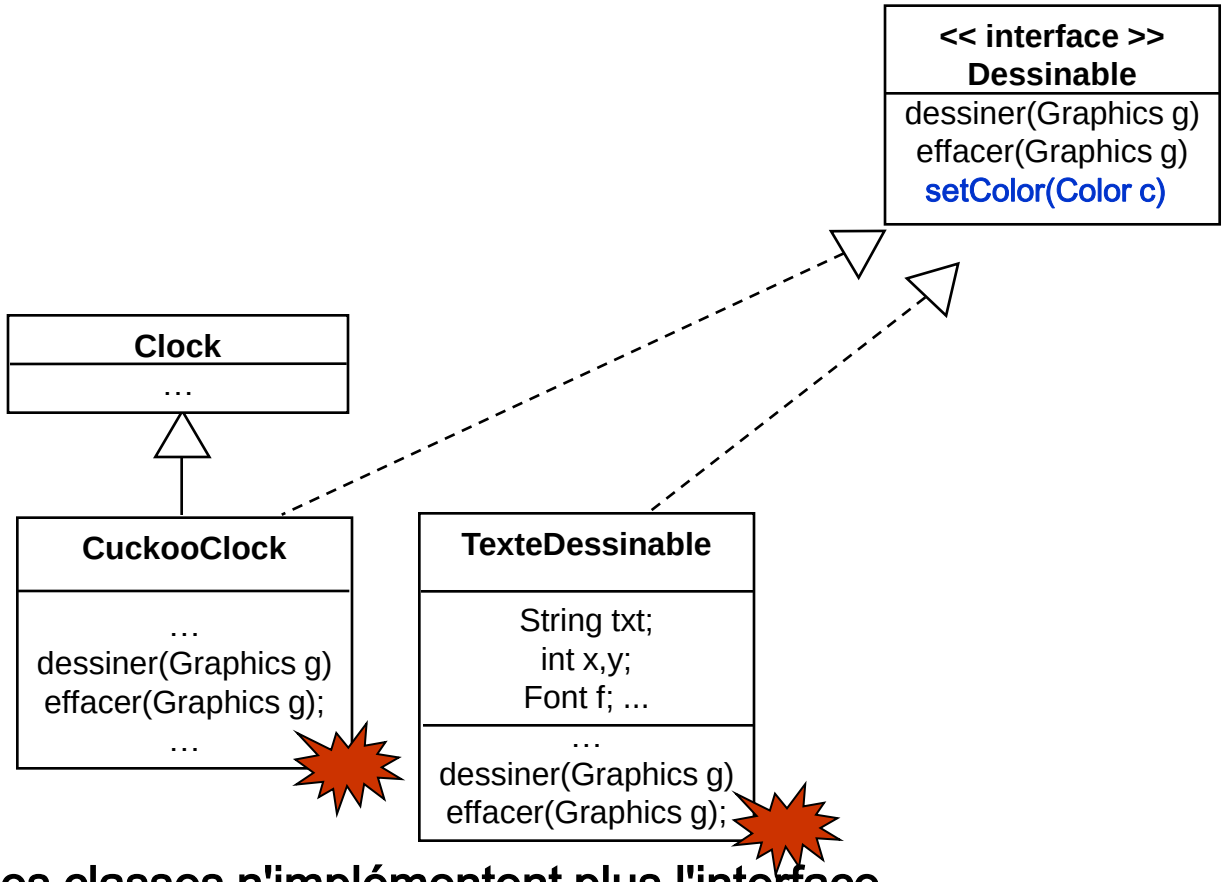
interêt

« Smarter Java development » *Michael Cymerman* , javaworld aout 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

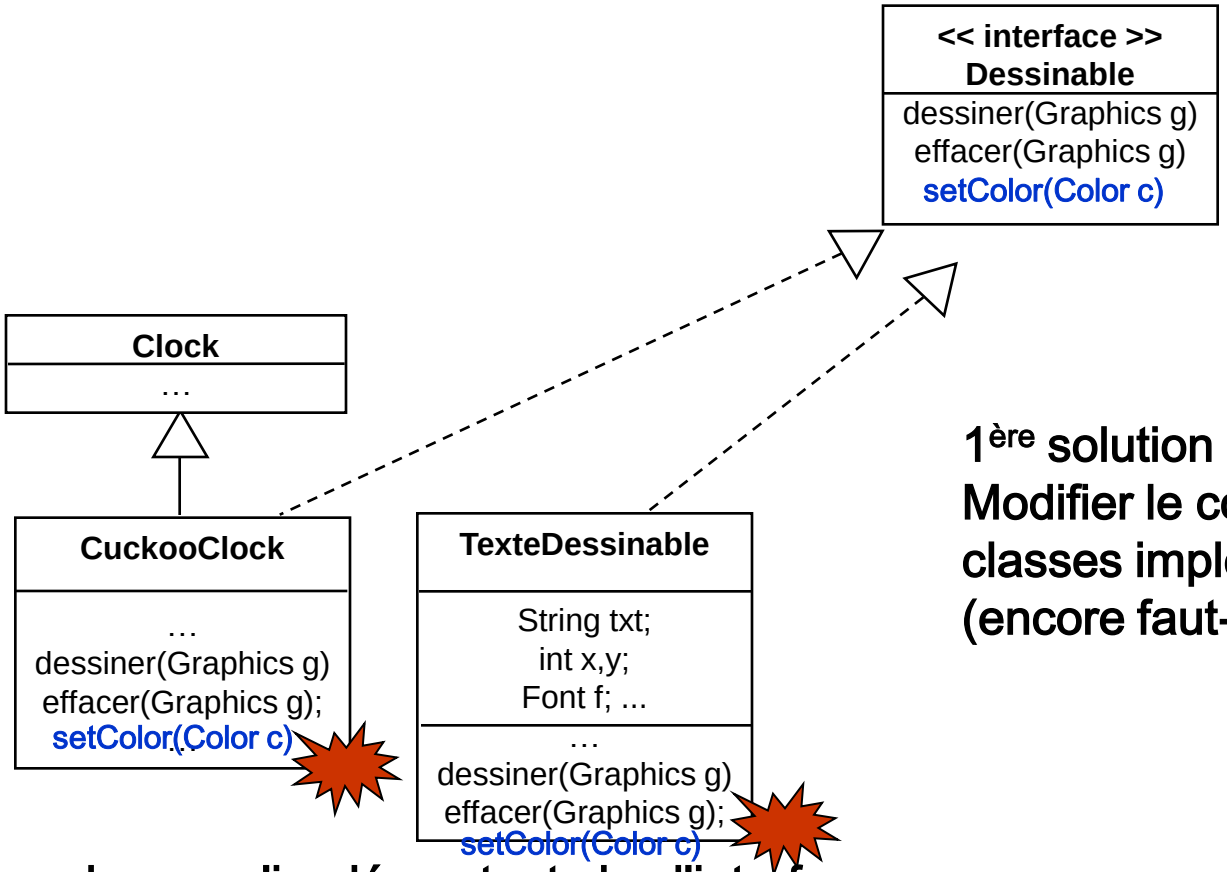
- Quand on définit des interfaces il faut être prudent : tout ajout ultérieur "brise" le code des classes qui implémentent l'interface.



Ces classes n'implémentent plus l'interface

Evolution des interfaces

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

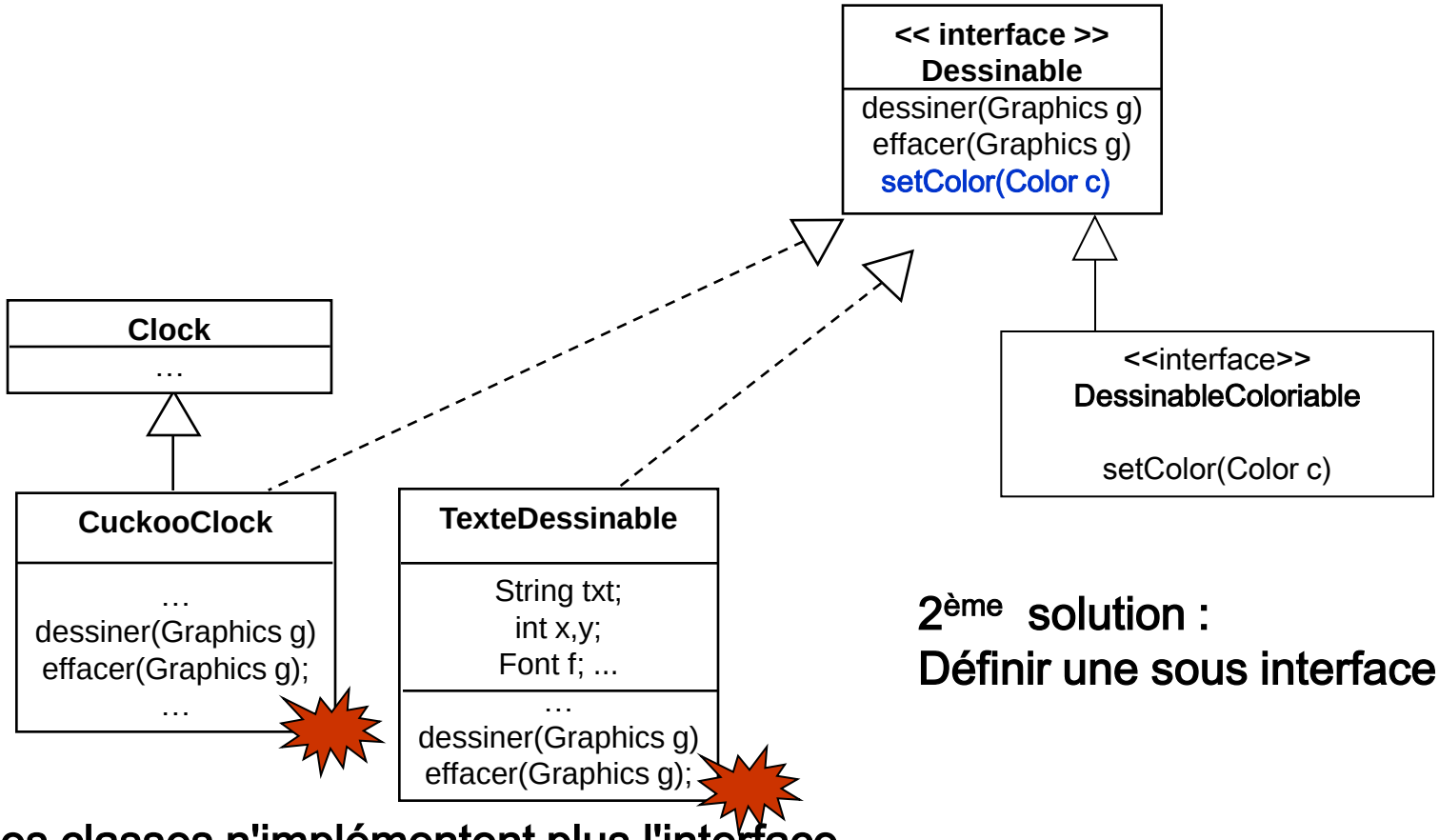


1ère solution :
Modifier le code de toutes les classes implémentant l'interface (encore faut-il le pouvoir)

Ces classes n'implémentent plus l'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.



2ème solution :
Définir une sous interface

Ces classes n'implémentent plus l'interface