# Bases de Java — Typage et liaison dynamique

LU3IN002 : Programmation par objets

L3, Sorbonne Université

`https://moodle-sciences-23.sorbonne-universite.fr`

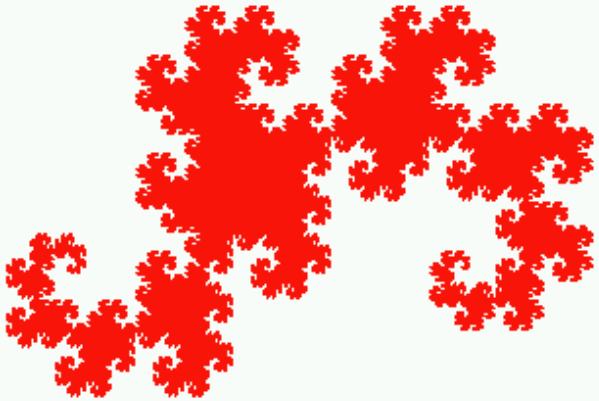
---

Antoine Miné

Cours 3

20 septembre 2023

Année 2023–2024



SORBONNE UNIVERSITÉ

LIP6

# Plan du cours

- Cours 1, 2 & 3 : Introduction et bases de Java
- Cours 4 : Collections, itérateurs
- Cours 5 : Exceptions, tests unitaires
- Cours 6 : Design patterns I : Design Patterns structurels
- Cours 7 : Polymorphisme
- Cours 8 : Design patterns II : Design Patterns comportementaux
- Cours 9 : Interfaces graphiques (JavaFX)
- Cours 10 : Design patterns III : Design Patterns créationnels
- Cours 11 : Aspects fonctionnels de Java, lambdas

Aujourd'hui :

- typage
- liaison dynamique (ou liaison tardive)
- classes abstraites
- premiers *design patterns*

# Typage

# Type statique, type dynamique

**Type statique :** déclaré dans le code source

Java est un langage statiquement et fortement typé

$\Longrightarrow$ tout attribut, variable locale, argument est déclaré avec un type

le type offre des garanties sur le comportement à l'exécution

Le type statique est immuable, connu à la compilation :

- type primitif (`int`, `double`, etc.) ;
- classe ou interface ;
- tableau de classes ou d'interfaces.

**Type dynamique :** de l'objet réellement référencé à l'exécution

Durant l'exécution, une variable de type classe `C` peut contenir `null`,
ou une référence vers une instance de `C`, ou d'une classe dont `C` est un ancêtre.

- défini par l'instruction `new` qui a créé l'objet référencé (type classe)
- évolue lors de l'exécution (à chaque affectation)

Exemple : une variable de type `Rectangle` peut référencer un objet de type `RoundedRectangle` à un moment, puis un objet de type `Rectangle` à un autre.

# Conversion implicite entre types statiques

La conversion du type statique vers une classe ancêtre est silencieuse et implicite :

- affectation dans une variable ou un attribut
- passage en argument à une méthode
- appel à une méthode héritée.

exemples

```
void f(Rectangle r) { r.draw(); ... }

RoundedRectangle x = new RoundedRectangle(...);

f(x);                    // f( (Rectangle) x )
Rectangle y = x;         // Rectangle y = (Rectangle) x
x.getWidth();            // ((Rectangle) x).getWidth()
```

- `x` a pour type statique et type dynamique `RoundedRectangle`
- `r` et `y` ont pour type statique `Rectangle`
  et pour type dynamique `RoundedRectangle`
- la conversion ne modifie pas l'objet référencé !

Ces conversions implicites sont toujours sûres ! (pas d'erreur de type possible à l'exécution)

$\Longrightarrow$ `f`, programmé vis à vis de `Rectangle`, accepte aussi des `RoundedRectangle`

# Conversion explicite entre types statiques

Java s'assure qu'une variable de type statique `A`
contient toujours un objet de classe `A` ou d'une classe dérivée de `A`.

Le type statique `A` contraint à accéder uniquement aux méthodes et attributs de `A`
même si la variable référence une instance de classe dérivée de `A`.

Pour accéder aux attributs et méthodes ajoutées par une classe dérivée,
il est nécessaire de convertir explicitement le type statique.

**Opérateur de conversion :** `(C) expr`

Si la classe `C` n'est pas un ancêtre du type dynamique de `expr`,
la conversion échoue avec l'erreur `InvalidCastException`.

La conversion ne modifie pas l'objet, elle donne juste une information supplémentaire sur son type.

exemple

```
void f(Rectangle r) {
   RoundedRectangle g = (RoundedRectangle) r;
   g.getRadius(); // accès possible via g, mais pas via r
}
f(new RoundedRectangle(...));    // OK
f(new Rectangle(...);            // echec dans f
```

# Notion théorique : le sous-typage

Si la classe `B` est un ancêtre de la classe `A`
on dit que le type `A` est un sous-type de `B`, noté `A <: B`.

Exemple : `RoundedRectangle <: Rectangle <: Object`.

**Règle de substitution de Liskov**

Si `A <: B`, un objet de type `A` peut être utilisé là où un objet de type `B` est attendu.

Java autorise donc la conversion implicite (sans test dynamique ni risque d'erreur)
d'une expression de type statique `A` en une expression de type statique `B` ancêtre de `A`.

Justification :

- une expression de type statique `A` contient un objet d'un type dérivé de `A` ;
- l'objet est donc également d'un type dérivé de `B`, ancêtre de `A` ;
- le type dynamique de l'objet a bien tous les attributs et méthodes de `B`.

D'autres langages ont d'autre relations de sous-typage permettant la réutilisation du code avec un type différent sans test dynamique : e.g., sous-typage structurel...

# Test dynamique de type : `getClass`

Le type dynamique d'un objet à l'exécution peut être trouvé par introspection :

- Java crée pour chaque classe une instance de la classe `java.lang.Class` qui donne des détails sur sa définition (nom, attributs, méthodes, etc.) ;
- la méthode standard `Class getClass()` d'`Object` retourne l'instance de `Class` correspondant à la classe de l'objet appelant ;
- chaque classe a un attribut statique `class` de type `Class` ;
- la méthode statique `Class.forName(String)` trouve une instance de `Class` à partir du nom d'une classe.

Les instances de `Class` peuvent être alors :

- transformées en chaîne : `getName()`
- comparées : `==` (une seule instance pour chaque classe)
- inspectées pour lister les attributs, les méthodes, etc.

exemples

```
Object o = ...;
if (o.getClass() == String.class) ...
if (o.getClass() == Class.forName("String")) ...
System.out.println(o.getClass().getName());
```

# Test dynamique de type : `instanceof`

Plutôt que de connaître la classe exacte d'un objet
il est souvent plus utile de savoir s'il est d'une classe donnée ou dérivée

i.e., s'il peut être converti en une classe donnée sans `InvalidCastException`.

**Test de compatibilité de type :** `expr instanceof C`

- vrai si et seulement si `C` est un ancêtre de la classe de `expr`
  i.e., vrai si et seulement si `(C) expr` n'échoue pas

  $\Longrightarrow$ toujours protéger les conversions par `instanceof` !
- n'a de sens que si le type statique de `expr` est un ancêtre de `C`.

exemple

```
void f(Rectangle r) {
   if (r instanceof RoundedRectangle) {
       RoundedRectangle g = (RoundedRectangle) r;
       g.getRadius();
   }
}
```

Depuis Java 14, on peut écrire plus simplement :

```
if (r instanceof RoundedRectangle g) { g.getRadius() }
```

# Application : méthode `equals` polymorphe

L'opérateur `==` effectue un test d'égalité physique (de référence) sur les objets.

Parfois, une égalité de contenu (structurelle) est plus utile.

$\Longrightarrow$ nous utilisons la méthode standard `equals`
par défaut, identique à ==, mais pouvant être redéfinie.

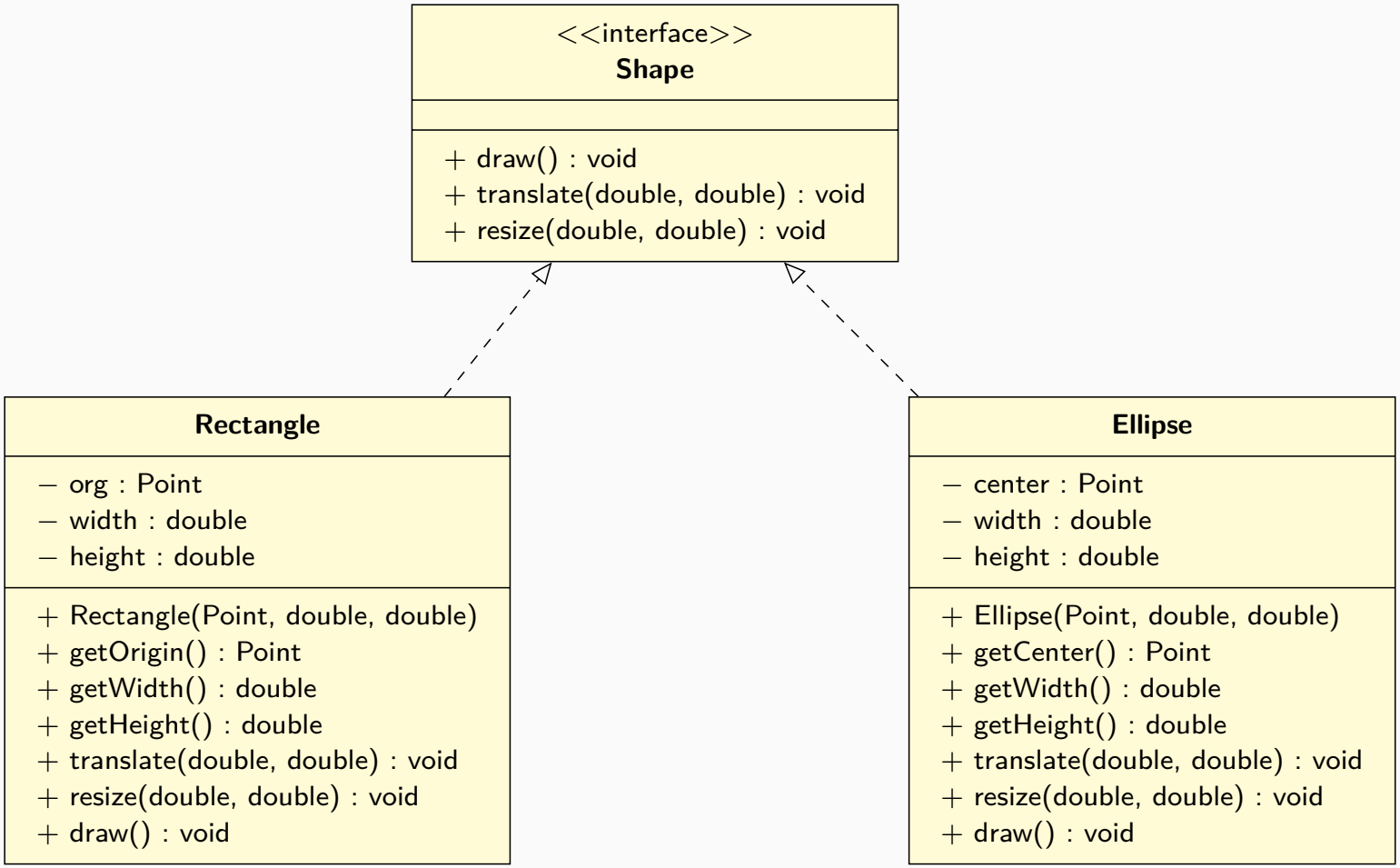
`public boolean equals(Object o)`

- asymétrique : compare `this` à un autre objet ;
- prend un argument de type `Object` pour être générique ;
- souvent, seul comparer des objets classes compatibles a un sens
  et il faut accéder aux attributs de o
  $\Longrightarrow$ utilisation d'un test dynamique de classe et d'une conversion.

exemple

```
public class Point {
    @Override public boolean equals(Object o) {
        if (!(o instanceof Point)) return false;
        Point p = (Point) o;
        return (x == p.x) && (y == p.y); // pas o.x ni o.y
    }
}
```

# Interfaces et implantations



# Interfaces comme types

Il est impossible de créer un objet de type dynamique interface

`new Shape(...)` est illégal (pas de constructeur).

rappel : `Shape`, vue au cours 2, est une interface implantée par `Rectangle` et `RoundedRectangle`

Mais une interface est un type statique autorisé ; il est possible de :

- déclarer une variable, un argument, un attribut de type interface : `Shape r;`
- vérifier si un objet implante une interface donnée : `r instanceof Shape`
- convertir une expression en un type interface donné : `(Shape) r`

  la conversion échoue à l'exécution si `r instanceof Shape` n'est pas vrai

**Sûreté du typage :**

Une variable `r` de type statique `Shape` contiendra à l'exécution
toujours une référence vers une instance d'une classe obéissant à l'interface `Shape`.

$\Longrightarrow$ toute méthode de l'interface peut être appelée sur `r`.

```
Shape r = new Rectangle(...);
r.draw();
```

# Sous-typage des tableaux

## Typage des tableaux :

`C[] var` signifie : tout élément de `var` référence un objet de classe `C` ou dérivée.

on peut aussi faire un tableau d'interfaces : `Shape[] shapes`

Principe de covariance : si `A <: B` alors, `A[] <: B[]`

Si `B` est un parent de `A`, la conversion de `A[]` en `B[]` est implicite et acceptée.

```
RoundedRectangle[] x = new RoundedRectangle[99];
x[0] = new RoundedRectangle(...);
Rectangle[] y = x;
```

- une lecture `y[0]` renvoie bien un objet dérivé de `Rectangle`

  $\Longrightarrow$ le type statique `Rectangle` est correct

- une écriture `y[0] = new Rectangle()` est, par contre, **incorrecte**

  `x` contiendrait un `Rectangle`, malgré son type `RoundedRectangle[]` !

La vérification statique de type est insuffisante à la sûreté de l'écriture ;

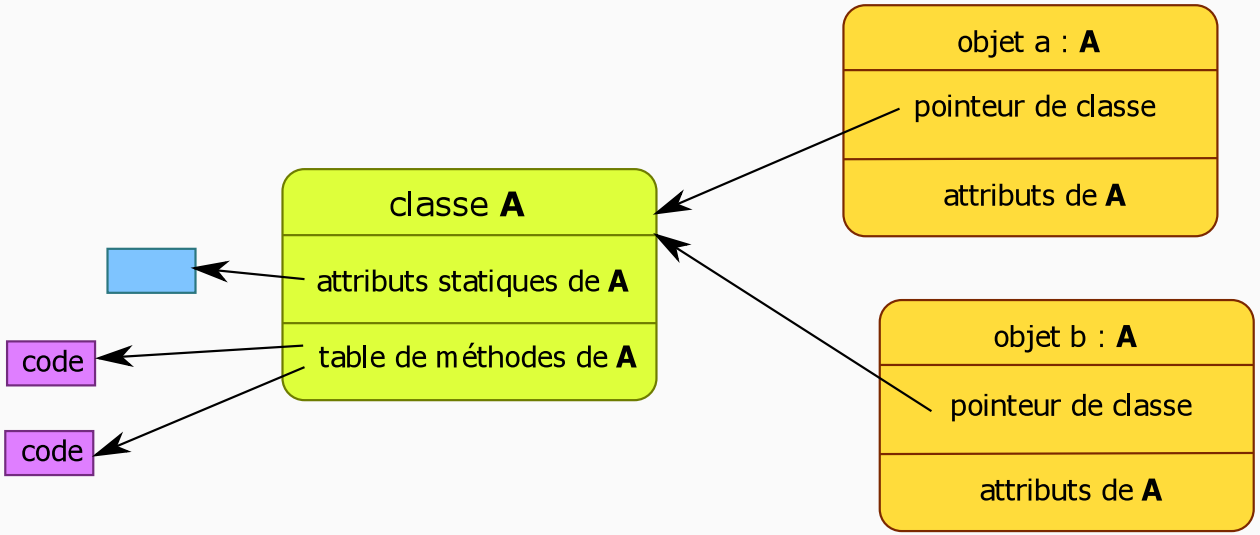
Java insère une vérification dynamique à chaque écriture dans un tableau !

$\Longrightarrow$ `ArrayStoreException` signalée en cas d'erreur

lent, et génère des erreurs de type à l'exécution au lieu de la compilation...

# Héritage et liaison dynamique

# Représentation dynamique des classes et des objets (1/2)



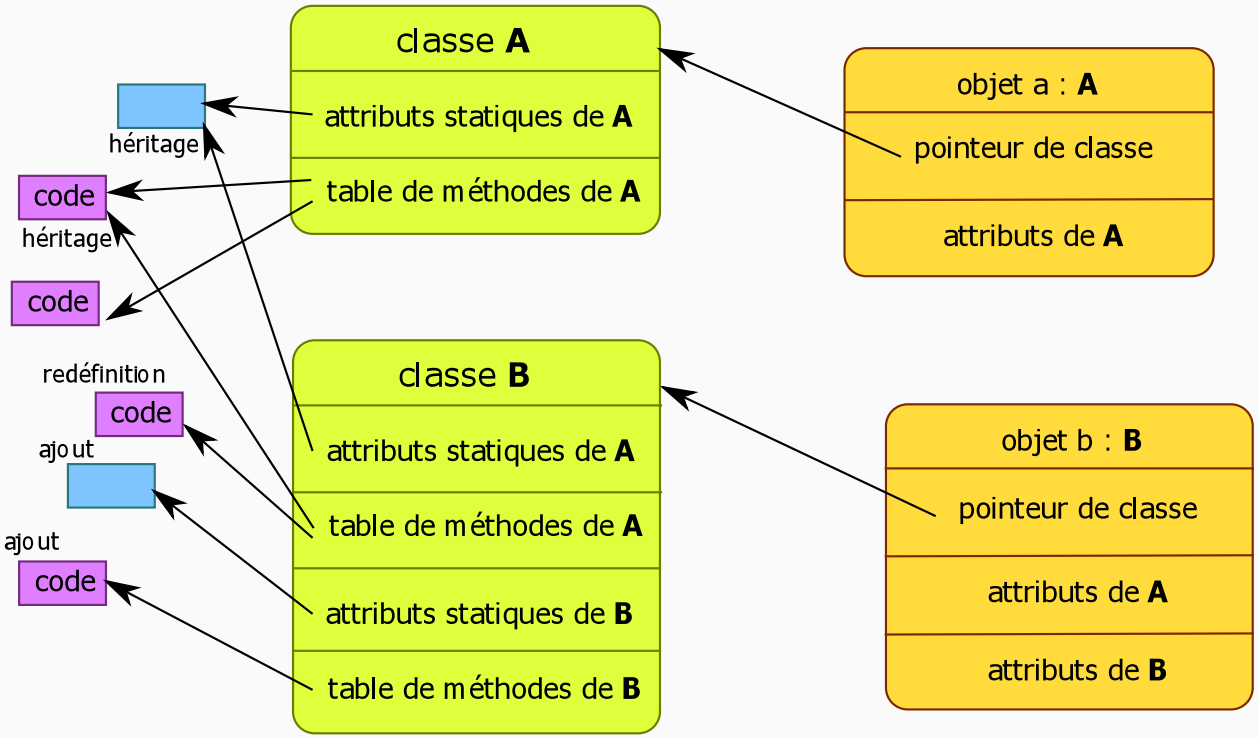

Une structure de classe contient l'information commune à tous les objets :

- la valeur des attributs statiques ;
- les pointeurs vers le code des méthodes.

Une structure d'objet contient :

- un pointeur vers la classe (type dynamique) ;
- la valeur des attributs non-statiques, spécifiques à chaque instance.

# Représentation dynamique des classes et des objets (2/2)




Supposons « `B extends A` ». Par principe de substitution :

- la représentation d'un objet de classe `B` est compatible avec (étend) celle de la classe `A` ;
- la représentation de la structure de classe `B` est compatible avec la classe `A` ;
  même attributs, mêmes méthodes aux mêmes indices dans les tables
- les attributs statiques sont hérités ou ajoutés ;
- les méthodes sont héritées, redéfinies, ou ajoutées.

# Liaison dynamique

fournisseur

```
public class A {
  public void a() ...
  public void b() { a(); }
}
public class B extends A {
  @Override public void a() ...
}
```

client

```
A x = new B();
x.a(); // appelle B::a()
x.b(); // appelle A::b(), qui appelle B::a()
```

Lors d'un appel de méthode, le **type dynamique** est toujours utilisé pour déterminer le code de la méthode effectivement appelée,

y compris lors d'un appel sur `this` au sein d'une classe !

Le type statique sert seulement à déterminer que la méthode existe, mais pas quel code sera exécuté.

Avantage :

- mécanisme puissant de polymorphisme
  une classe dérivée peut altérer le comportement de son parent.

Inconvénients :

- légèrement plus coûteux qu'un appel résolu statiquement ;
  indirection supplémentaire via la table de méthodes de la structure de classe pointée par l'objet
- très difficile de déterminer *de visu* quel code est effectivement exécuté.

# Exemple d'utilisation de la liaison dynamique

pobj/cours3/Rectangle.java

```
public class Rectangle {

  public void draw() {
    drawHLine(org, width);
    ...
  }

  void drawHLine(Point p, double width) ...
  void drawVLine(Point p, double width) ...
}
```

pobj/cours3/ThickRectangle.java

```
public class ThickRectangle
extends Rectangle {

  @Override
  void drawHLine(Point p, double width) ...

  @Override
  void drawVLine(Point p, double width) ...
}
```

Un rectangle épais `ThickRectangle` est une nouvelle sorte de `Rectangle`.

Soit : `ThickRectangle r = new ThickRectangle(...); r.draw();`

- `ThickRectangle` ne redéfinit pas `draw`
  $\Longrightarrow$ c'est le code `draw` de `Rectangle` qui est utilisé ;
- `ThickRectangle` redéfinit `drawHLine`
  $\Longrightarrow$ le code de `draw` appelle la nouvelle fonction `drawHLine`.

# Exemple d'utilisation de la liaison dynamique (client)

test

```
class Test {

  private void drawGrille(Rectangle r) {
    for (int i=0; i<10; i++) {
        for (int j=0; j<10; j++) {
          r.draw();
          r.translate(10,0);
        }
        r.translate(-100,10);
    }
    r.translate(0,-100);
  }

  public static void Main(String[] args) {
    Test test = new Test();
    test.drawGrille(new ThickRectangle(new Point(), 10, 10));
  }
}
```

Le type dynamique `ThickRectangle` est utilisé,
et non le type statique `Rectangle`.

La méthode pour afficher une grille affiche maintenant une grille épaisse...

Puissant, mais dangereux !

# Redéfinition et type de retour

Rappel : une méthode est redéfinie si elle a le même nom et les mêmes types d'argument que dans le parent, sinon c'est une nouvelle définition.

Dans une redéfinition, le type de retour doit :

- soit être identique au type indiqué dans la méthode du parent ;
- soit être un sous-type (classe héritée, donc plus précise).

Le type statique lors de l'appel détermine la signature utilisée
et donc le type statique de retour.

La méthode réellement appelée peut naturelle retourner un objet d'un sous-type...

copie d'objet

```
public class A {
  @Override
  public A clone() { ... }
}

public class B extends A {
  @Override
  public B clone() { ... }
}
```

client

```
B b = new B();
b.clone(); // type statique et dynamique : B

A a = new B();
a.clone(); // type statique : A, type dynamique : B

// dans les deux cas B::clone est appelée

B z = a.clone(); // erreur de type, ne compile pas !
```

# Visibilité et redéfinition

# Redéfinition de méthodes privées

Rappel : une méthode d'une classe n'est pas visible dans une sous-classe si :

- elle est de visibilité `private` ;
- ou de visibilité package et la sous-classe est dans un package différent.

Dans ce cas, la méthode :

- ne peut pas être appelée directement par la sous-classe ;
- ne peut pas être redéfinie (liaison statique, non dynamique).

Tout définition d'une méthode de même nom sera une nouvelle définition, donc jamais appelée par la classe parent.

**fournisseur**

```
public class A {
    private void x() ...
    public void y() { x(); }
}

public class B extends A {
    // nouvelle définition
    private void x() ...
}
```

**client**

```
B b = new B();
b.x(); // appelle B::x()
b.y(); // appelle A::y(), qui appelle A::x()
```

# Liaison statique, redéfinition de méthodes statiques

L'opposé de la liaison dynamique (tardive)
est la liaison statique (précoce), résolue à la compilation,
en utilisant le type statique.

En Java, la liaison statique est utilisée pour les méthodes statiques (ou privées).

en C++, la liaison est statique par défaut pour toutes les méthodes,
et la liaison dynamique doit être choisie explicitement avec le mot-clé `virtual`

Une définition de méthode statique est toujours une nouvelle définition.

Elle masque l'ancienne définition,
mais seulement pour les variables de type statique hérité !

bibliothèque

```
public class A {
  public static void s() ...
  public void a() { s(); }
}

public class B extends A {
  // nouvelle définition
  public static void s() ...
}
```

client

```
B.s(); // appelle B::s()

A b = new B();
b.s(); // appelle A::s()
       // car b a pour type statique A

b.a(); // appelle A::a(), qui appelle A::s()
```

# Redéfinition d'attributs

La redéfinition de méthodes dans une classe dérivée est très utile.

La redéfinition d'un attribut est autorisée, mais peu utile.

Une redéfinition masque l'attribut de la classe parent
$\Longrightarrow$ les deux attributs coexistent dans la représentation de l'objet.

La résolution de l'attribut se fait statiquement, pas dynamiquement !

exemple

```
class A {
  public int a;
  public int getA() { return a; }
}
A x = new A();
x.a;      // A::a
x.getA(); // A::a
```

exemple

```
class B extends A {
  public int a;
  public int getA() {
    // retourne B::a car this a type statique B
    return a;
  }
}
A x = new B();
x.a;      // A::a, car x a pour type statique A
x.getA(); // B::a, car B::getA() est appelée
```

Note : la question ne se pose pas si vous définissez tous les attributs comme privés,
il s'agit forcément d'une nouvelle définition car les attributs privés ne sont pas hérités !

Si les attributs ne sont accédés que par des méthodes d'accès,
les règles de la liaison dynamique de méthodes s'appliquent $\Longrightarrow$ fortement conseillé !

# Méthodes `final`

Problème : le contrôle d'accès ne fait pas la distinction entre :

- une méthode pouvant être appelée par une sous-classe ;
- une méthode pouvant être redéfinie par une sous-classe.

Or, en redéfinissant une méthode publique, une sous-classe peut briser l'encapsulation !

bibliothèque

```
class Login {
  public String enterPassword() {
    // masque la saisie du mot de passe
    // en affichant des *
  }
}
```

attaquant

```
class Crack extends Login {
    public String enterPassword() {
       String s = super.enterPassword();
       System.out.println(s);
       return s;
    }
}
```

Exemple : en passant au client une instance de `Crack` au lieu de `Login`, le mot de passe saisi devient visible !

**Solution :** mot-clé `final` pour une méthode

```
public final String enterPassword()
```

interdit la redéfinition de la méthode dans toute sous-classe

tout en permettant de l'appeler, car la méthode est `public`

# Classes `final`

exemple

```
package java.lang;
public final class String {
    ...
}
```

Une classe définie `final` ne peut pas être étendue par héritage.

toutes ses méthodes sont donc implicitement `final`

Très utilisé par les classes standards Java de `java.lang`
pour assurer la robustesse et la sécurité des codes client.

Évite les erreurs ou hacks par redéfinition du comportement d'une classe de base.

Exemples :

- `System` : accès au système (horloge, console, I/O) ;
- `String` : chaînes de caractères immuables ;
- `Integer`, `Boolean`, `Double`, ...
  versions « classe » des types primitifs.
  utiles pour stocker ces valeurs dans un conteneur ou avoir une représentation uniforme dérivé de `Object`

L'extension de ces classes pourra se faire par composition, plutôt que par héritage.

# Classes internes

Gallery.java

```
public class Gallery {
   private String name;
   private List<Painting> paintings = ...;

   private class Painting {
       private String title;
       public Painting(String t) { title = t; }
       public String toString() { return name + ":" + title; }
   }

   public void addPainting(String title) { paintings.add(new Painting(title)); }
 }
```

Une classe interne (ici, `Painting`) :

- est définie dans une classe parent (ici, `Gallery`) ;
- peut être privée (invisible, inaccessible, sauf pour le parent)
  ou publique (accessible de l'extérieur avec la notation `.`, comme `Gallery.Painting`).

Une instance de la classe interne :

- est créée par une instance du parent,
- garde une référence vers l'objet parent,
- et peut accéder aux tous ses attributs et méthodes, même privés.
  (relation privilégiée, évite d'exposer trop au monde extérieur, plus en cours 4 et 7)

# Classes abstraites

# Recherche d'un ancêtre commun pour les classes formes

Problématique : toutes les formes ont des caractéristiques communes :

- des propriétés communes : position et taille ;
- des opérations communes : déplacer, afficher.

Nous souhaitons faire dériver toutes nos classes formes
d'un ancêtre commun `AbstractShape`, pour :

- factoriser les définitions d'attributs communs ;
- factoriser le code des méthodes indépendantes du type de forme ;
  (e.g., accesseurs, `translate`, `resize`)
- en gardant non-définies les méthodes qui dépendent du type de forme
  (e.g, `draw`)

Comme `Rectangle`, `Ellipse`, etc., `AbstractShape` obéit à l'interface `Shape`,
mais elle ne fournit qu'une implantation partielle.

Elle ne doit pas être instanciée directement,
mais seulement héritée par des classes qui complètement l'implantation.

# Mauvaise idée : une classe

pobj/cours3/AbstractShape.java

```
public class AbstractShape implements Shape {
  private Point origin;
  private double width, height;

  public AbstractShape(Point origin, double width, double height)
  {  this.origin = origin; this.width = width; this.height = height; }

  public Point getOrigin() { return origin; }
  public double getWidth() { return width; }
  public double getHeight() { return height; }

  public void translate(double x, double y) { origin.translate(x,y); draw(); }
  public void resize(double w, double h) { width = w; height = h; draw(); }

  public void draw() { System.out.println("Not Implemented. Use a subclass!"); }
}
```

La méthode `draw` doit être présente car elle est :

- demandée par `implements Shape` ;
- et surtout, appelée directement par `translate` et `resize` !

nous fournissons une implantation « vide », à redéfinir dans les classes héritées.

Danger : aucune protection contre `x = new AbstractShape(...); x.draw();`

# Solution : classes abstraites, mot-clé `abstract`

**Classes abstraites :** nous voulons

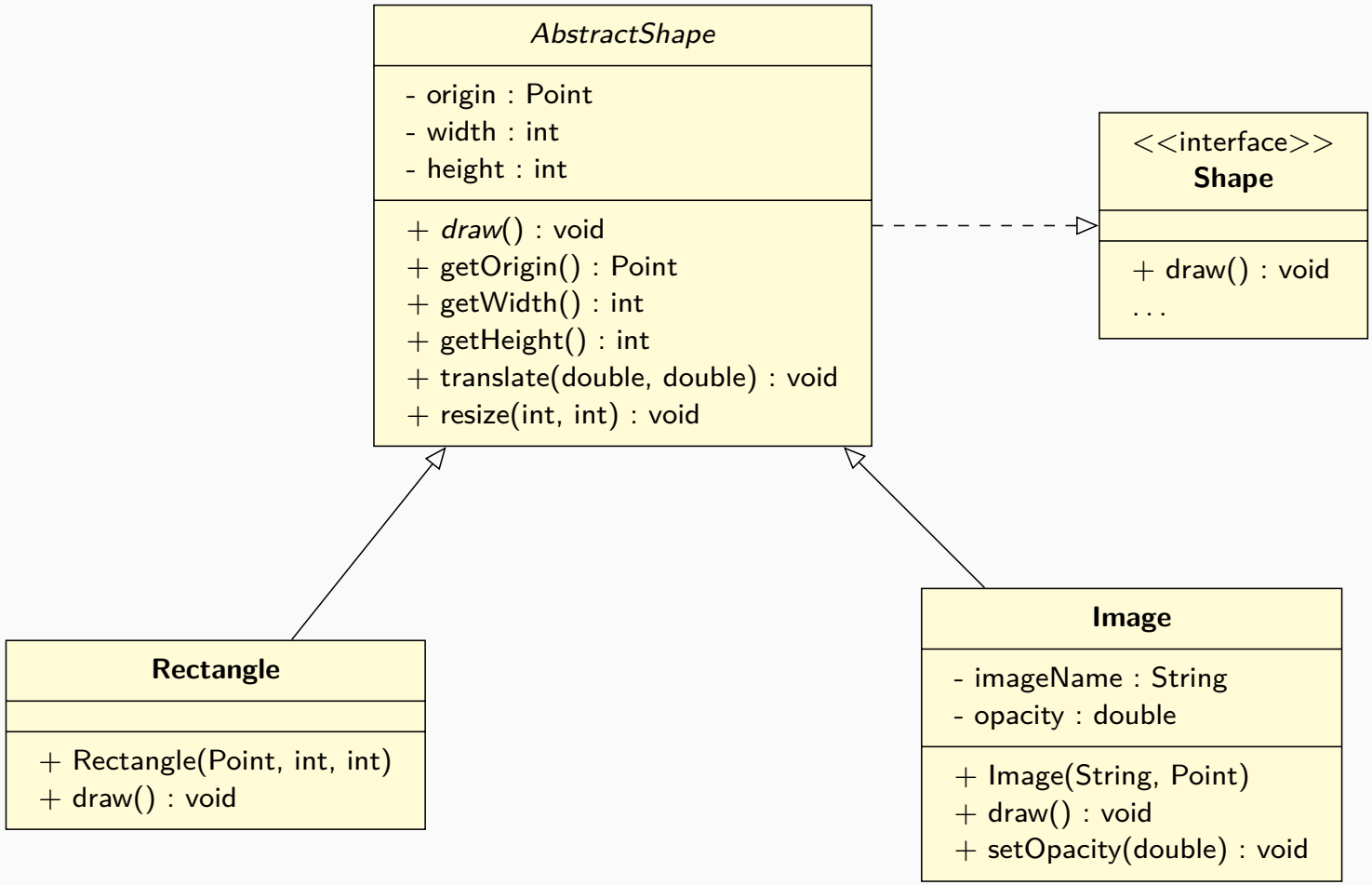
- éviter que la classe `AbstractShape` soit instanciée par erreur ;
  $\Longrightarrow$ modificateur `abstract` dans la déclaration de la classe :
- ne pas fournir d'implantation de `draw` (même une implantation vide),
  mais forcer toute classe dérivée instanciable à redéfinir `draw` ;
  $\Longrightarrow$ modificateur `abstract` dans la déclaration de la méthode.

`abstract` signale ces intentions, qui seront vérifiées par le compilateur.

pobj/cours3/AbstractShape.java

```
abstract public class AbstractShape implements Shape {
  private Point origin;
  private double width, height;

  public AbstractShape(Point origin, double width, double height)
  {  this.origin = origin; this.width = width; this.height = height; }

  public Point getOrigin() { return origin; }
  public double getWidth() { return width; }
  public double getHeight() { return height; }

  public void translate(double x, double y) { origin.translate(x,y); draw(); }
  public void resize(double w, double h) { width = w; height = h; draw(); }

  abstract public void draw();
}
```

# Classes abstraites : diagrammes UML




- Les classes abstraites sont indiquées en *italique*.
- Les méthodes abstraites sont indiquées en *italique*.
- Une classe ayant au moins une méthode abstraite est forcément abstraite,
  mais on peut déclarer abstraite une classe complètement définie, pour interdire son instanciation...

# Résumé : classes, classes abstraites et interfaces

| | classes | classes abstraites | interfaces |
|---|---|---|---|
| instanciable | oui | non | non |
| héritage | simple | simple | multiple |
| constructeurs | oui | oui | non |
| attributs | oui | oui | non |
| méthodes avec code | oui | oui | oui (*) |
| méthodes sans code | non | oui | oui |
| méthodes privées | oui | oui | oui (**) |

- classe : implantation complète, utilisable ;
- classe abstraite : implantation incomplète, à sous-classer ;
  $\Longrightarrow$ but : factorisation d'une implantation
- interface : vue publique, contrat
  $\Longrightarrow$ but : indépendance vis à vis d'une implantation.

(*) méthodes publiques `default` et `static` ajoutées aux interfaces en Java 8, cf transparent suivant
(**) méthodes privées et privées `static` ajoutées aux interfaces en Java 9
Ces ajouts augmentent la réutilisabilité du code, sans changer la liaison dynamique.

# Méthodes `default` dans les interfaces

Java 8 introduit les méthodes `default`,

pour spécifier une implantation des méthodes d'interface.

une méthode `default` peut bien sûr être redéfinie dans une classe implantant la méthode

Ajoute à Java des notions de :

- héritage multiple ;

  avec résolution statique des ambiguïtés :
  impossible d'hériter de deux interfaces ayant une même méthode avec implantation `default`

- mixins (mécanismes de composition de classes)

Mais les interfaces Java ne peuvent toujours pas définir d'attribut, ni de constructeur.

exemple : interface de types ordonnés

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

public interface Ordered<T> extends Comparable<T>
{
    default public boolean equal(T obj)   { return compareTo(obj) == 0; }
    default public boolean less(T obj)    { return compareTo(obj) < 0; }
    default public boolean greater(T obj) { return compareTo(obj) > 0; }
}
```

# Introduction aux design patterns

# Motivation

Observation empirique (par « GoF » : Gamma, Helm, Johnson, Vlissides, en 94) que
les solutions robustes à des problèmes courants de conception logicielle
orientée objet gravitent autour d'un petit nombre de motifs souvent réutilisés

$\Longrightarrow$ il est utile de les cataloguer.

Un *design pattern* est :

- une solution réutilisable à un problème d'ingénierie logicielle courant
  ni un algorithme, ni une bibliothèque, ni un trait simple de langage
- un ensemble de classes et d'interfaces, et leurs relations ($\Rightarrow$ UML)
- observé dans plusieurs applications (généralité de la solution)
- nommé, avec des rôles bien identifiés pour les classes et les interfaces qui le composent
  important pour communiquer au sein d'une équipe de développement
- à connaître, pour éviter de tâtonner et de réinventer (mal) la roue
- à utiliser avec discernement !

# Héritage vs. composition : un rectangle escamotable

dépendance sur l'implantation

```
public class RectangleOpt
extends Rectangle {

  private boolean show = true;

  public RectangleOpt(...) {
    super(...);
  }

  @Override public void draw()
  { if (show) super.draw(); }
}
```

dépendance sur l'interface

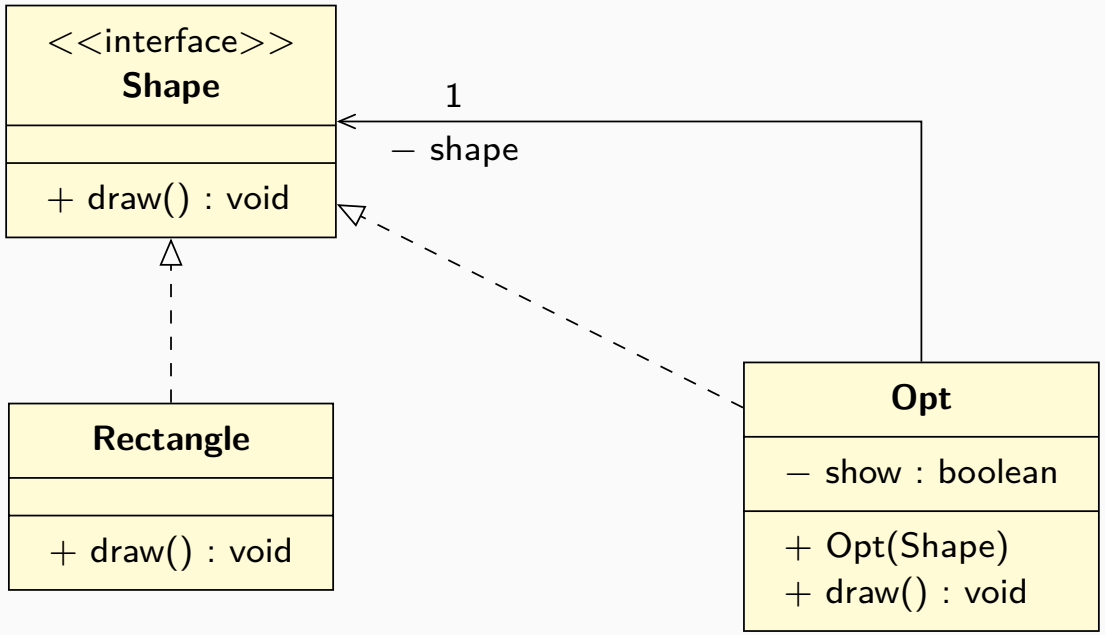
```
public class Opt implements Shape {

  private Shape shape;
  private boolean show = true;

  public Opt(Shape s) {
  { shape = p; }

  static Opt newRectangleOpt(...) {
    Rectangle r = new Rectangle(...);
    return new Opt(r);
  }

  public void draw()
  { if (show) shape.draw(); }

  public void translate(double x, double y)
  { shape.translate(x,y); }

  public void resize(double w, double h)
  { shape.resize(w,h); }
}
```

## Variante de rectangle : `RectangleOpt`

La version de gauche est plus courte, mais limitée aux rectangles.

La version de droite est généralisable à d'autres formes (interface `Shape`).

# Diagramme UML de la composition

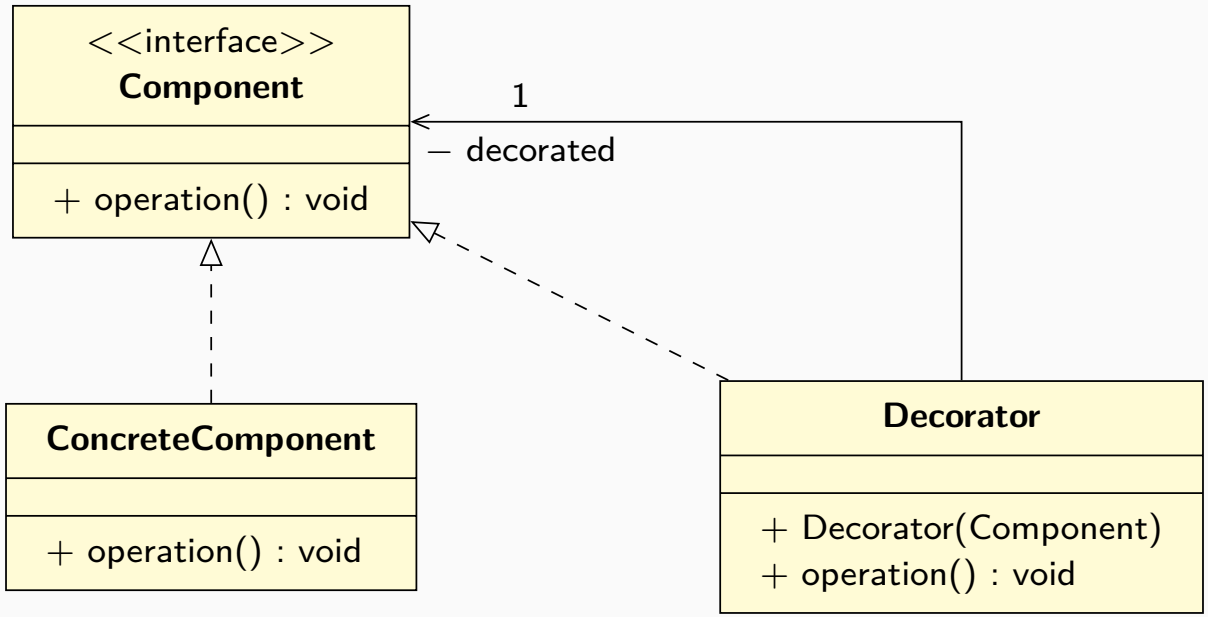



- `Shape` est l'interface des objets affichables (`draw`)
- `Rectangle` est un rectangle simple, implantant `Shape`
- `Opt` permet de rendre une forme « escamotable » (`show`)

$\Longrightarrow$ `Opt` implante et délègue à la même interface `Shape`.

C'est le *design pattern* Décorateur.

# Généralisation : motif Décorateur

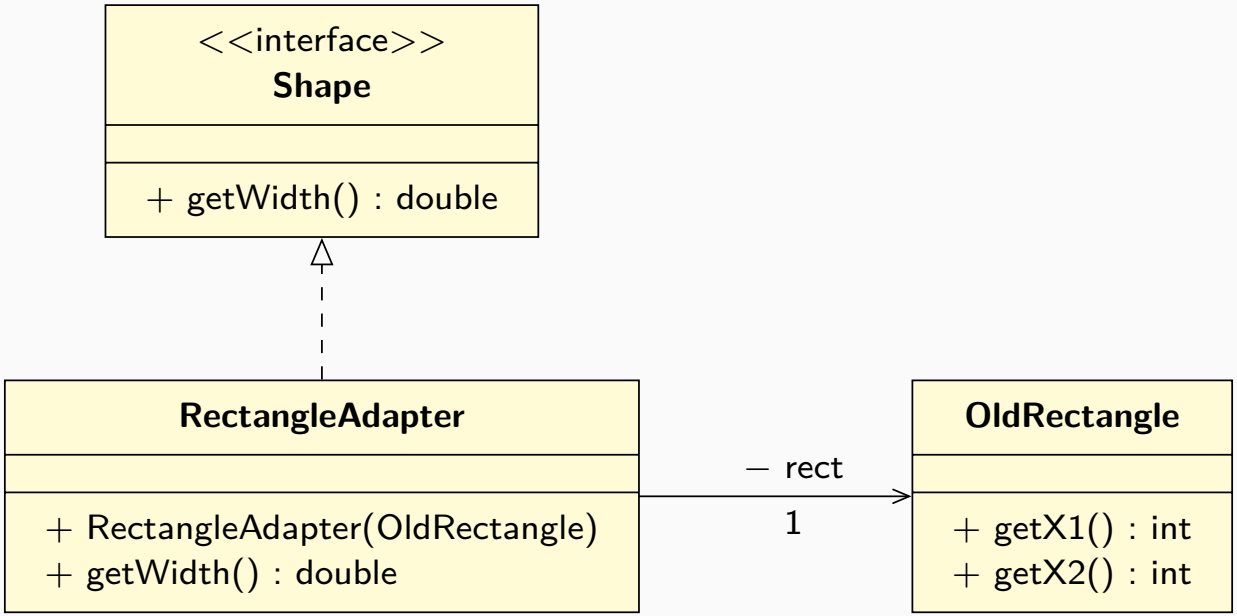



Fonction du *design pattern* Décorateur :
ajout d'une responsabilité sans modifier la hiérarchie de classes.

Autres exemples de décorateurs :

- `java.io.BufferedReader` décore `java.io.Reader`
- ajout de cache ou de proxy
- débogage et log

# Motif Adaptateur




RectangleAdapter.java

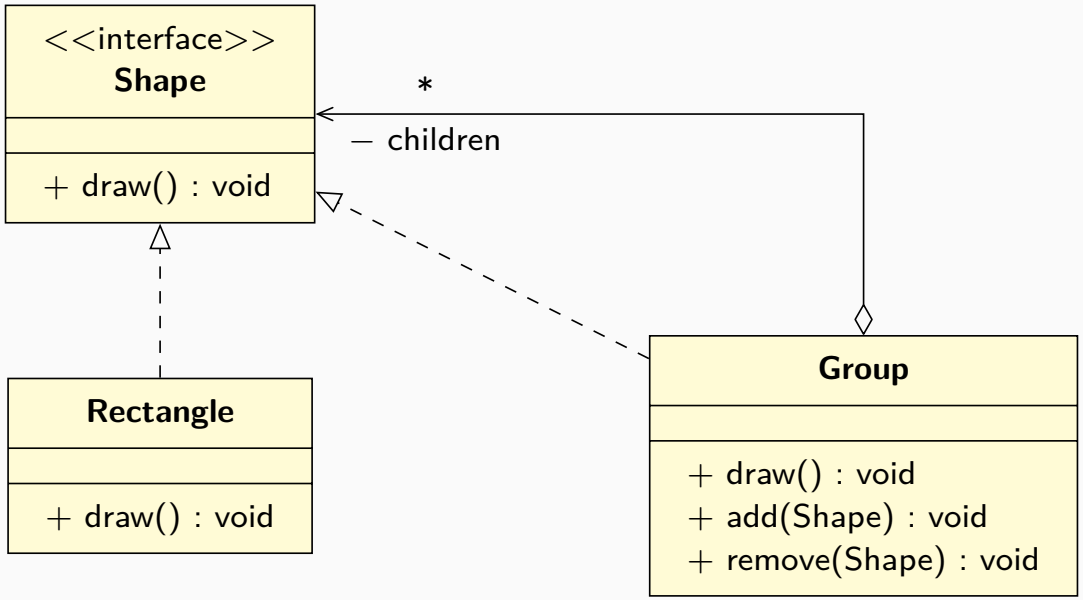
```
public class RectangleAdapter implements Shape {
    private OldRectangle rect;
    public RectangleAdapter(OldRectangle rect) { this.rect = rect; }
    public double getWidth() { return Math.abs(rect.getX2() - rect.getX1()); }
}
```

Fonction du *design pattern* Adaptateur :

convertir un objet d'une interface à une autre :

- référence un objet ayant une mauvaise interface ;
- délègue à cet objet les méthodes de l'interface correcte.

# Motif Composite




- `*` indique qu'un `Group` contient une collection de formes (`Shape`) ;
- les formes peuvent être ajoutées et enlevées d'un groupe à volonté ;

  ♢ indique que la durée de vie d'une forme n'est pas liée à celle du groupe ;
  une forme peut appartenir à plusieurs groupes, ou à aucun.
- un groupe est également une forme.

  Un groupe peut même contenir d'autres groupes !

C'est un nouvel exemple de *design pattern* : le motif **Composite**.

**Principe :** le client traite un groupe comme si c'était un élément simple.

# Implantation de composite

pobj/cours3/Group.java

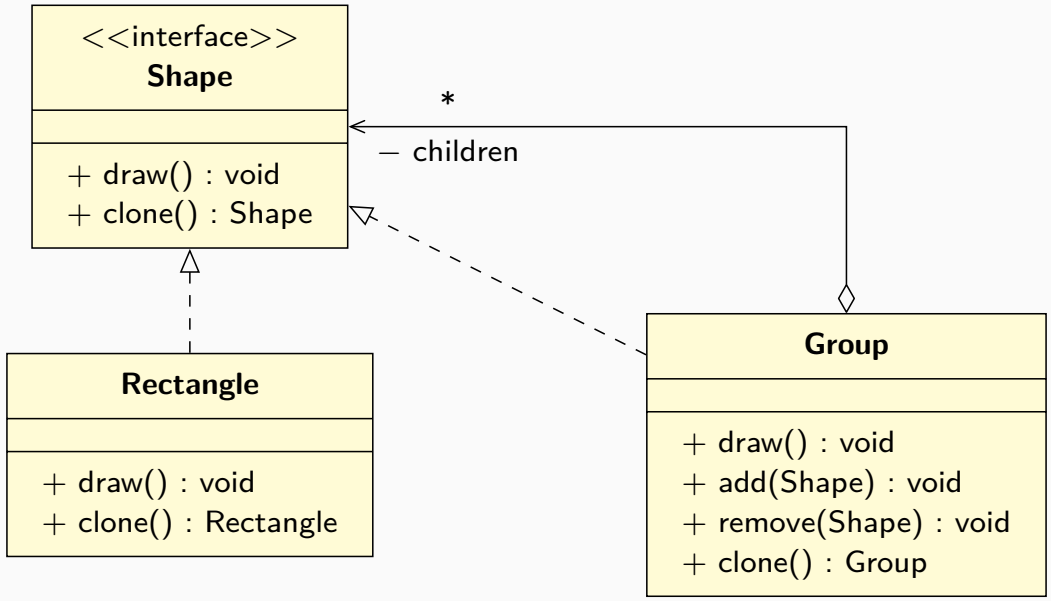
```
public class Group implements Shape {

private List<Shape> children = new LinkedList<Shape>();

    public Group()                { }

    public void add(Shape s)      { children.add(s); }
    public void remove(Shape s)   { children.remove(s); }

    public void draw() {
        for (Shape s : children)
            s.draw();
    }
}
```

Les objets `Shape` forment un arbre.

Dessiner un groupe va parcourir l'arbre récursivement
pour dessiner toutes les formes qu'il contient aux feuilles (e.g., `Rectangle`).

Organisation très extensible grâce à l'implantation vis à vis d'une interface et à la liaison dynamique !

# Copie de composite : retour sur la liaison dynamique




pobj/cours3/Group.java

```
public class Group implements Shape {
    public Group clone() {
        Group d = new Group();
        for (Shape s : children) d.add(s.clone());
        return d;
    }
}
```

Copie récursive dans le cas des groupes de formes.

La liaison dynamique assure que la bonne méthode de copie est toujours appelée.