Bases de Java — Héritage

3I002 : Programmation par objets
L3, UPMC

http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2015/ue/3I002-2016fev/

Antoine Miné

Année 2015–2016

Cours 2
26 janvier 2016

Plan du cours

- Cours 1, 2 & 3 : Introduction et bases de Java
- Cours 4 : Introduction aux *design patterns*
- Cours 5 : Exceptions, tests (JUnit)
- Cours 6 : Polymorphisme, surcharge et génériques
- Cours 7 : Interfaces graphiques
- Cours 8 : Collections
- Cours 9 : Entrées/sorties, réseau, sérialisation
- Cours 10 : Aspects fonctionnels de Java, révisions de Java
- Cours 11 : Génie logiciel, révision des *design patterns*

Aujourd'hui :

- héritage
- typage
- interfaces

Rappels : principes de la programmation orientée objet

Programmation robuste et extensible.

Grâce à des traits de langage, des bonnes pratiques de programmation et des briques réutilisables de conception logicielles (*design patterns*).

Principes :

1. encapsulation

   objets, classes, packages

2. abstraction

   contrôle d'accès, interfaces

3. réutilisabilité

   héritage, composition

4. polymorphisme

   typage, hiérarchie de classes, liaison tardive

Fil conducteur

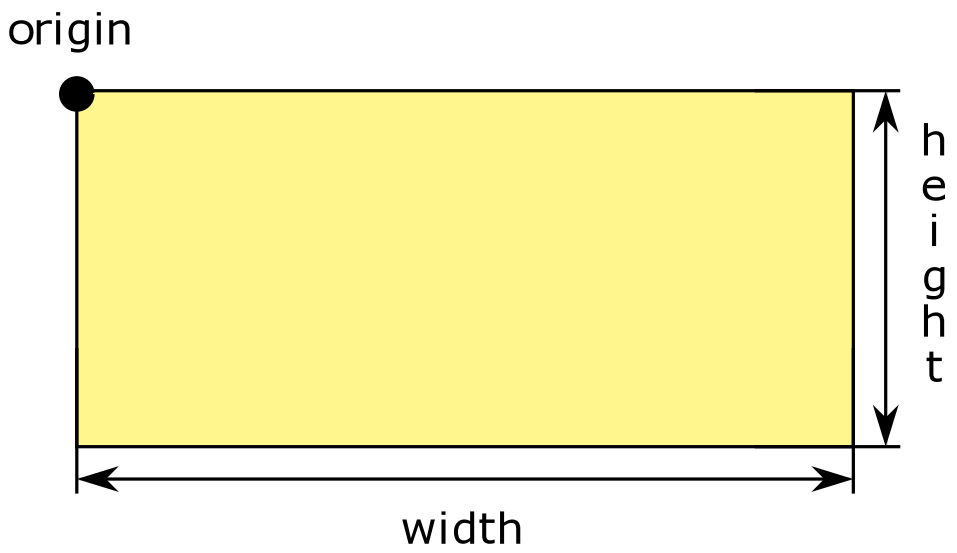
Nous illustrons les concepts sur une application simple : un logiciel de dessin.

Le logiciel manipule des formes (rectangles, carrés, etc.).

Les formes sont des objets avec :

- des attributs, privés (e.g. : position, taille, etc.) ;
- des invariants (e.g. : la taille est positive) ;
- des constructeurs pour créer ces objets ;
- des méthodes publiques pour modifier ces propriétés ;
  (en respectant toujours les invariants)
- des méthodes privées (ou visibles du package), utilisées en interne.

Exemple : le rectangle



Un rectangle est défini par :

- un point *origine* (coin supérieur gauche) ;
- une taille *width*, *height*.

Notes :

- La taille *width*, *height* est, comme l'origine, une paire de nombres, mais ce n'est pas un point ! Les méthodes d'un point n'auraient pas de sens sur ces données.
- Les tailles *width*, *height* doivent être toujours positives.
  Java n'a pas de type non-signé ; cela ne peut pas être exprimé par typage.
  $\Longrightarrow$ Ce sera donc un invariant, maintenu par nos méthodes.

Rappels : la classe en Java

pobj/cours2/Rectangle.java

```
package pobj.cours2;

import pobj.cours1.Point;

public class Rectangle {

  private Point org;
  private int width, height;

  public Rectangle(Point origin,
                   int width, int height) {
    org = origin;
    this.width = (width > 0) ? width : 1;
    this.height = (height > 0) ? height : 1;
  }

  public Point getOrigin() { return org; }
  public int getWidth()  { return width; }
  public int getHeight() { return height; }
```

pobj/cours2/Rectangle.java

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

  public void resize(int nWidth, nHeight) {
    if (nWidth <= 0 || nHeight <= 0) return;
      width = nWidth; height = nHeight;
  }

  public void draw() {
    drawHLine(org, width);
    ...
  }
  void drawHLine(Point p, int width) { ... }
  void drawVLine(Point p, int width) { ... }

  @Override public String toString() {
    return width + "x" + height + "@" + org;
  }
}
```

Nouvelle classe `Rectangle`.

Nous utilisons un nouveau package `pobj.cours2` pour éviter les conflits avec le cours 1, tout en réutilisant la classe `Point` que nous avions défini.

Rappels : diagrammes de classes UML

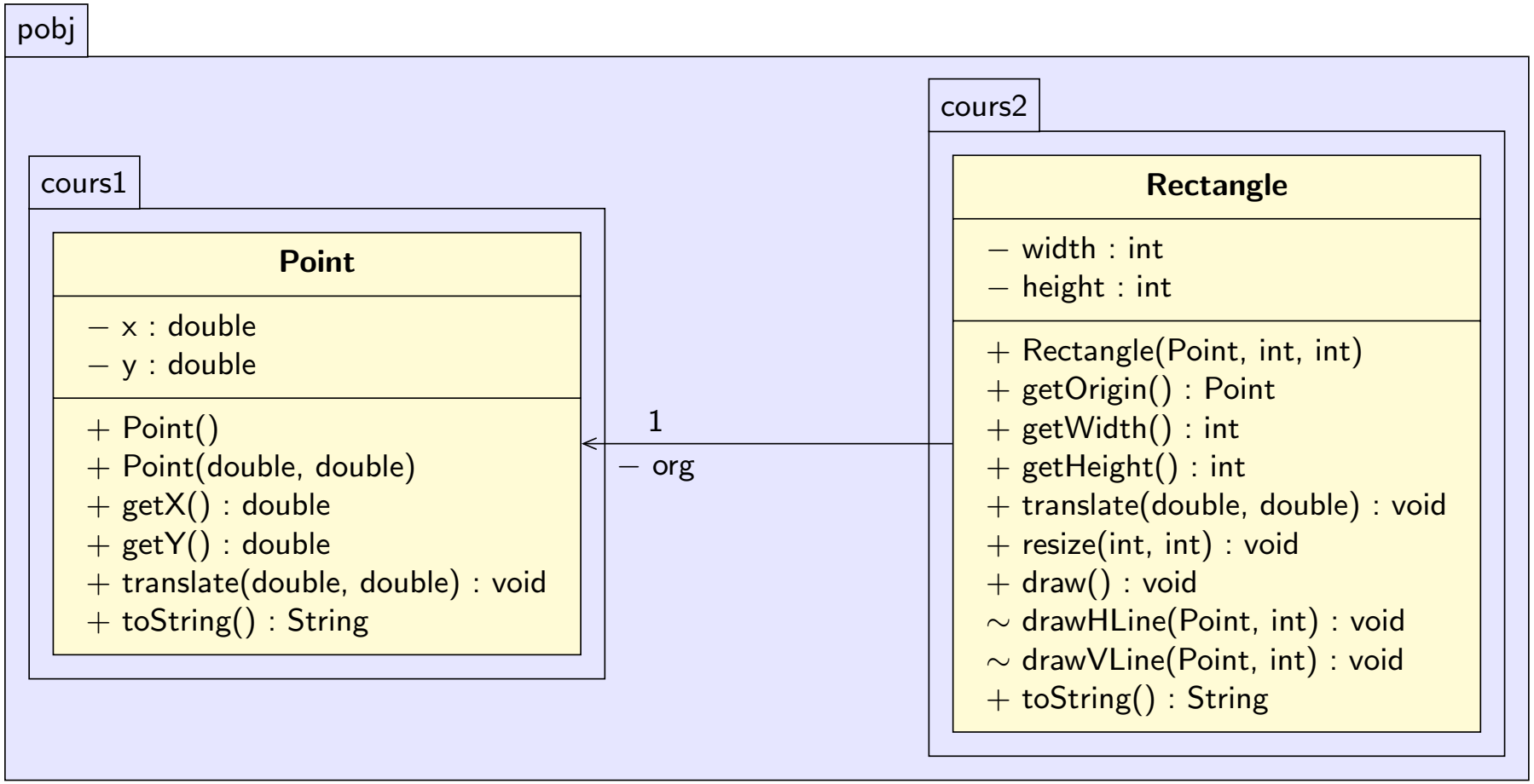


Diagramme UML montrant les classes, leur contenu et leurs relations :

- le diagramme peut faire apparaître les packages ;
- le diagramme peut omettre des attributs et méthodes (vue abstraite).

Le cycle de vie des objets

En Java, tout objet est une instance d'une classe.

1. Le programme créé un objet par un appel à `new`.
   e.g. : `new Rectangle(new Point (100, 100), 10, 20)`

   Le système :
   - charge la classe, si elle n'est pas déjà présente en mémoire ;
   - alloue de l'espace mémoire pour l'objet ;
   - initialise les attributs, et appelle le constructeur.

2. La référence sur le nouvel objet, renvoyée par `new` est stockée, copiée, passée en argument,... par le programme.

   Le programme accède aux attributs et appelle les méthodes de l'objet via une référence.

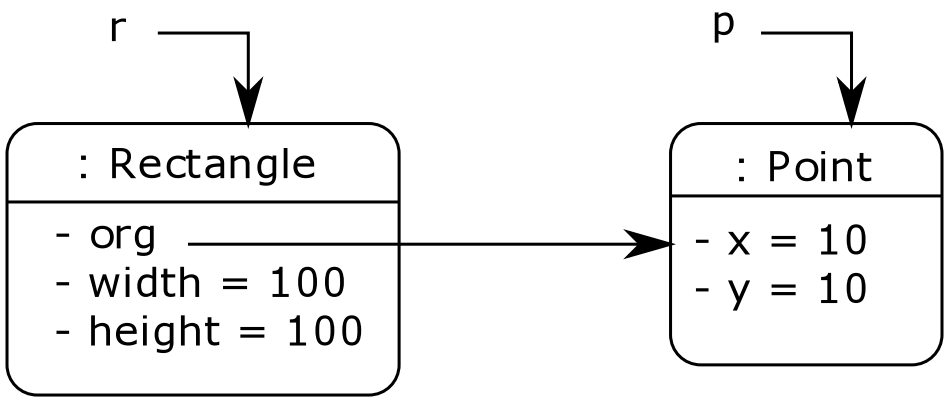
3. Le système libère **automatiquement** la mémoire quand l'objet n'est plus référencé grâce à un algorithme de *Garbage Collection* (GC)
   Pas de libération manuelle $\Longrightarrow$ pas d'erreur de gestion mémoire ;
   mais il faut pas garder de références sur un objet inutilisé !
   $\Longrightarrow$ penser à mettre à `null` les attributs des objets de longue vie si plus utiles.

4. Si elle est déclarée, la méthode `finalize` est appelée juste avant la libération, mais c'est rarement utile (la libération peut avoir lieu longtemps après la dernière utilisation).

Diagrammes d'objets UML

exemple

```
Point p = new Point(10, 10);
Rectangle r = new Rectangle(p, 100, 100);
```



Diagrammes d'objets : convention UML pour représenter :

- l'état de la mémoire à un point donné de l'exécution ;
- les instances d'objets existants, avec leur classe et la valeur des attributs ;
- les références entre objets ;
- les variables locales.

Méthode `clone`

L'affectation = copie une référence sur l'objet.

Il est parfois nécessaire de créer une copie de l'objet
qui sera manipulée de manière indépendante de l'original.

Java offre une méthode `protected Object clone()`, héritée depuis `Object`, mais elle est très difficile à exploiter !

Nous allons programmer notre propre méthode `clone`, ce qui permet de :

- choisir un type de retour plus précis que `Object` ;
- avoir le choix entre :
  - copie profonde : copie récursive des attributs ;
  - copie de surface : référence des attributs de l'original.

pobj/cours1/Point.java

```
public class Point implements Clonable {
  private double x,y;
  @Override public Point clone() {
    return new Point(x.y);
  }
}
```

pobj/cours2/Rectangle.java

```
public class Rectangle implements Clonable {
  private Point org;
  private int width, height;
  @Override public Rectangle clone() {
    Point p = org.clone(); // ou p = org
    return new Rectangle(p, width, height);
  }
}
```

(`Clonable` est une interface marqueur, voir plus loin)

Héritage

Principe de l'héritage ; mot-clé `extends`

Une classe peut hériter d'une (et une seule) autre classe,
en utilisant le mot-clé `extends`.

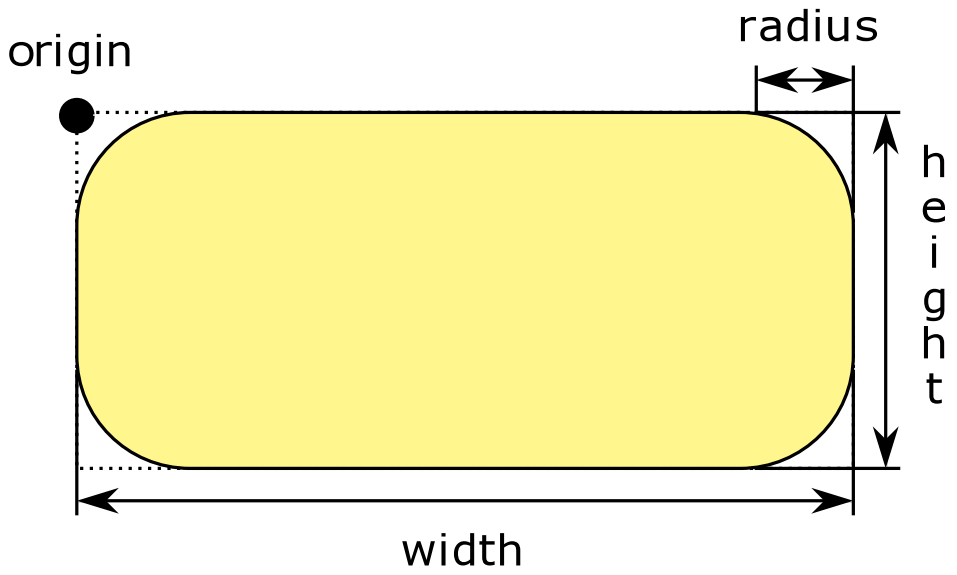
La classe dérivée peut :

- accéder aux méthodes et attributs de la classe parent ;
  (sous réserve de visibilité, voir plus loin)
- ajouter des nouveaux attributs à la classe parent ;
- redéfinir des méthodes de la classe parent ;

$\Longrightarrow$ la classe dérivée étend la classe parent.

L'héritage est un mécanisme fondamental de réutilisation de code
dans tous les langages à objets !

Exemple : les rectangle arrondi



Un rectangle aux coins arrondis est défini par :

- son origine et sa taille, comme un rectangle ;
- en plus : un rayon de courbure des coins.

Un rectangle peut être vu comme un rectangle arrondi où `radius` $= 0$
$\Longrightarrow$ le rectangle arrondi étend le rectangle.

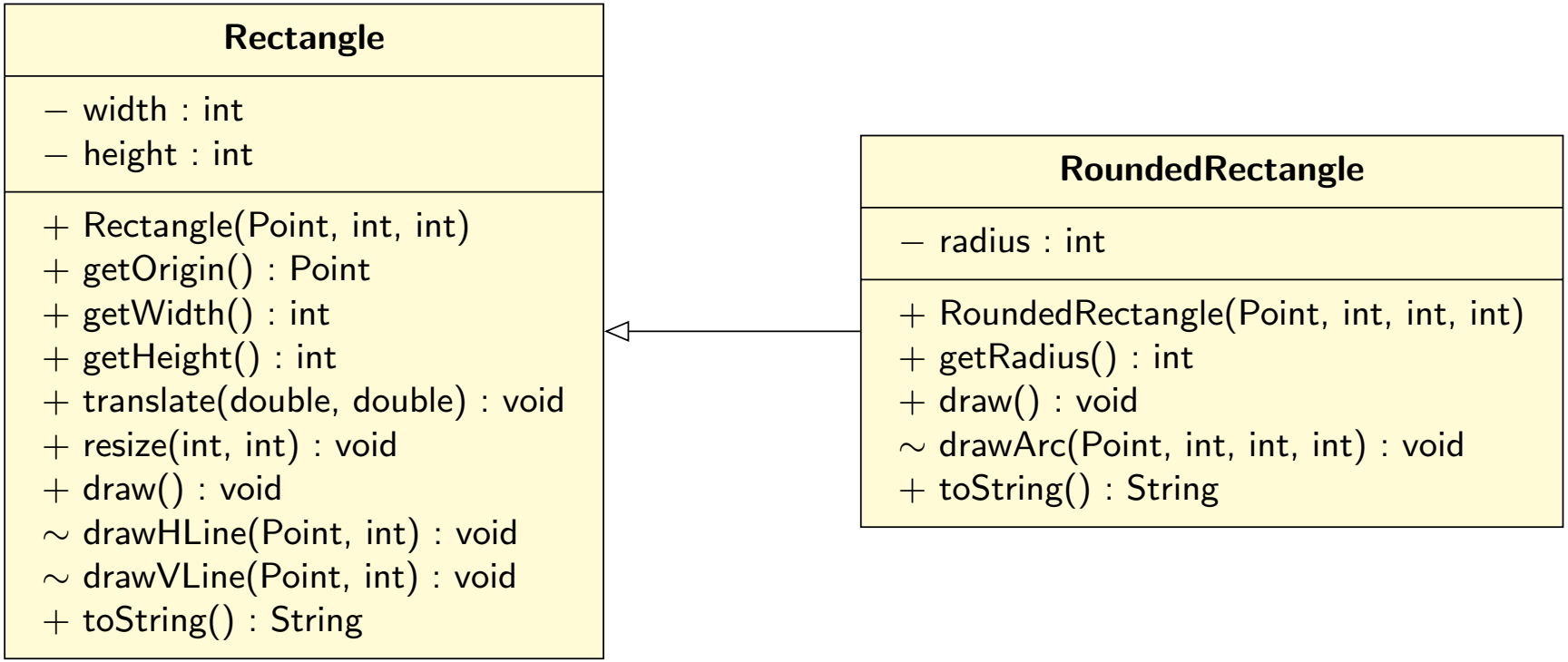
Nous voulons réutiliser au maximum le code de `Rectangle` existant.

Exemple d'héritage : les rectangles arrondis

pobj/cours2/RoundedRectangle.java

```
package pobj.cours2;

import pobj.cours1.Point;

public class RoundedRectangle extends Rectangle {

  private int radius;

  public int getRadius()  return radius;

  public RoundedRectangle(Point origin, int width, int height, int radius)
  { super(origin, width, height); this.radius = radius; }

  @Override public void draw() {
    drawHLine(...);
    drawArc(...);
    ...
  }

  void drawArc(Point c, int r, int angle1, int angle2) { ... }

  @Override public String toString()
  { return super.toString() + "/" + radius; }
}
```

Héritage en UML



- L'héritage est matérialisé par une association entre classes, avec une flèche creuse, de la classe dérivée vers le parent.
- La classe dérivée indique les attributs et méthodes ajoutés ou modifiés.
  les attributs et méthodes hérités restent implicites, comme dans le code Java

Accès aux attributs et méthodes hérités

pobj/cours2/RoundedRectangle.java

```
public class RoundedRectangle
extends Rectangle {
  ...
  @Override public void draw()
  { drawHLine(...); ... }
}
```

client

```
Point p = new Point();
RoundedRectangle r =
  new RoundedRectangle(p, 10, 20, 2);
int w = r.getWidth();
// accès à la méthode de Rectangle !
```

Une instance de classe dérivée possède, en mémoire,
tous les attributs d'une instance de classe parent.

Il est donc possible :

- d'accéder aux attributs déclarés dans la classe parent ;
- d'appeler une méthode de la classe parent,
  qui s'attend à la présence de ces attributs ;

sous réserve de respecter les règles de visibilité.

Visibilité

Visibilité des attributs, méthodes et constructeurs,
du plus permissif au plus strict :

UML	visibilité	mot-clé	accès autorisé pour : classe seule	toutes classes du package	toutes sous-classes	toutes classes
+	publique	`public`	✓	✓	✓	✓
#	protégée	`protected`	✓	✓	✓	
∼	package	pas de mot-clé	✓	✓		
−	privée	`private`	✓			

Rappel : un attribut privé peut rester accessible via une méthode publique !

Les classes dérivées ont tendance à exister dans le même package que leur parent, `protected` n'est donc pas très utilisé, et on lui préfère package ou `private`.

Une classe ne peut être que de visibilité publique ou package (par défaut)
$\Longrightarrow$ package permet de cacher la classe en dehors du package.

Enrichissement et redéfinition

pobj/cours2/RoundedRectangle.java

```
@Override public void draw() { ... drawArc(...); ... }
void drawArc(Point c, int r, int angle1, int angle2) { ... }
```

Une méthode de la classe héritée est :

- une redéfinition si elle a la même signature
  i.e., le même nom, et les mêmes types d'arguments ;
  par contre, le nom des arguments et le type de retour importent peu
- est une nouvelle définition sinon.

Exemples :

– `draw` est redéfini, car on ne dessine pas un rectangle arrondi comme on dessine un rectangle ;
– `getRadius` est ajouté car un rectangle n'avait pas de rayon ;
– `drawArc` est ajouté car un rectangle n'avait pas besoin de dessiner des arcs de cercles ;
– `getWidth` est hérité car la taille d'un rectangle arrondi a la même signification que la taille d'un rectangle.

L'intention de redéfinir une méthode est annoncée par `@Override` ;
optionnel, mais fortement conseillé pour bénéficier de vérifications dans Eclipse.

Lors de l'appel à la méthode `draw` sur un objet `RoundedRectangle`,
la nouvelle définition sera toujours utilisée : c'est la liaison dynamique.
(plus sur ce point au prochain cours)

Mot-clé super

pobj/cours2/RoundedRectangle.java

```
@Override public String toString()
{ return super.toString() + "/" + radius; }
```

Si une classe redéfinit une méthode du parent,
elle masque l'implantation du parent.

L'implantation du parent est toutefois accessible par le mot-clé `super` :

- `super.méthode(arg1, ..., argN)`
- limité à un seul niveau (`super.super` est interdit)
- limité au parent de la classe courante (`a.super` est interdit)
- fort parallèle avec le mot-clé `this`

Utilité principale : redéfinir une méthode du parent pour la classe dérivée
en adaptant aux nouveaux attributs, aux nouvelles fonctionnalités

(utilité aussi pour les constructeurs, voir le transparent suivant)

Héritage et constructeurs : mots-clés `super` et `this`

pobj/cours2/RoundedRectangle.java

```
public RoundedRectangle(Point origin, int width, int height, int radius)
{ super(origin, width, height); this.radius = radius; }

public RoundedRectangle(Point origin, int width, int height)
{ this(origin, width, height, 10); }
```

La classe dérivée n'hérite jamais des constructeurs du parent.
$\Longrightarrow$ il est nécessaire de redéfinir tous les constructeurs !

Justification :

Les objets de la classe dérivée ont des attributs supplémentaires, des invariants différents.
De nouveaux constructeurs sont nécessaires pour initialiser correctement l'état des objets.

Un constructeur de la classe peut commencer par appeler :

- `super(...)` pour appeler un constructeur de la classe parent
  $\Longrightarrow$ obligatoire pour initialiser des attributs privés du parent !
- `this(...)` pour appeler un constructeur de la classe en cours de définition
  $\Longrightarrow$ utile pour éviter de dupliquer du code d'initialisation
  c'est pour cela qu'un constructeur peut être privé !

Le constructeur ne peut utiliser qu'un seul des deux mots-clés, et c'est forcément sa première instruction.

Initialisation des attributs

Rappel : un nouvel objet a tous ses attributs initialisés

- soit explicitement par le constructeur ;
- soit explicitement par un initialiseur dans la déclaration de l'attribut ;
  `type attribut = expr;`
  `expr` peut même contenir un appel de méthode !
- soit explicitement, par un bloc d'initalisation,
  recopié par Java dans tous les constructeurs ;
- sinon, implicitement, à une valeur par défaut (0, 0.0, `false`, `null`)

Note : les variables locales ne sont pas automatiquement initialisées
et c'est une erreur d'oublier des les initialiser !

exemple

```
public class Toto {
    public Toto() { b = 2*4; }
    private int a = 2*4;  // initialisé à 12
    private int b;        // initialisé à 14
    private int c;        // initialisé à 0
    private int d;        // initialisé à 42
    { d = 42; }           // bloc d'initialisation
}
```

Constructeur par défaut

Constructeur par défaut

Java définit automatiquement un constructeur sans argument (initialisation par défaut) mais uniquement si la classe ne définit aucun constructeur.

Classe dérivée

Un constructeur de la classe parent est toujours appelé :

- si le constructeur ne commence pas par `super`,
  Java appelle automatiquement le constructeur sans argument `super()` ;
- erreur de compilation si ce constructeur n'existe pas !

exemple

```
public class A {
  public A(int x) { }
}

public class B extends A {
  public B(int x) { }            // erreur
  public B()      { super(0); }  // OK
}
```

Héritage et attributs `static`, `final`

Rappels :

- un attribut `final` est constant après initialisation par le constructeur ;
- un attribut `static` est partagé par toutes les instances d'une classe.

Un attribut static est aussi partagé par toutes les classes héritées !

exemple

```
public class Counter {
    static final public int max = 100;
    static private int nb = 0;
    private int val;

    public Counter() {
        val = nb;
        if (nb < max) nb++;
    }
}
```

exemple

```
public class NamedCounter
extends Counter {
    private String name;

    public NamedCounter(String name) {
        this.name = name;
    }
}
```

Un seul compteur pour toutes les instances de `Counter` et de `NamedCounter`...

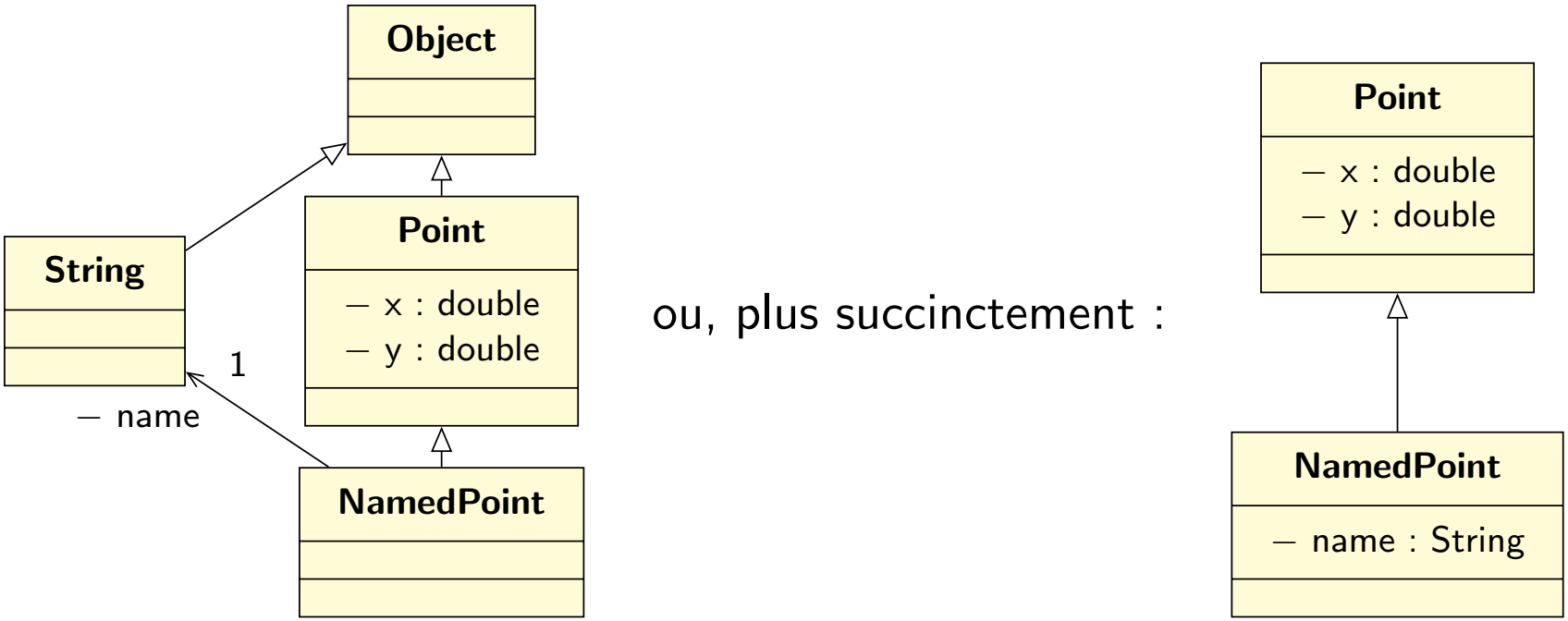
Note : utilisation de `static final` pour définir une constante symbolique globale.

Hiérarchie de classe, classe `Object`

Si `extends` n'est pas utilisé, la classe hérite automatiquement de `Object`.
$\Longrightarrow$ les classes forment un arbre enraciné à `Object`.

Les méthodes prédéfinies `toString`, `equals`, `clone`, etc.
sont simplement les méthodes héritées de la classe `Object` !

Exemple de hiérarchie : points et points nommés.



Les classes bien connues, comme `Object` ou `String`, sont souvent omises.
Les attributs immuables, comme `String`, sont souvent représentées par des attributs simples, sans association.

Typage

Type statique, type dynamique

Compilation

Java est un langage strictement typé
$\Longrightarrow$ tout attribut, variable locale, argument est déclaré avec un type.

Il s'agit du **type statique**, immuable, connu à la compilation.

Rappel : le type peut être

- un type primitif (`int`, `double`, etc.) ;
- une classe ;
- un tableau d'une classe.

Exécution

Durant l'exécution une variable type `Classe` peut contenir `null`,
ou une référence vers une instance d'une classe dont `Classe` est un ancêtre.
(la relation "est ancêtre de" est la clôture transitive de la relation "est parent de")

C'est le **type dynamique**, qui peut de plus évoluer durant l'exécution.

Exemple : une variable de type `Rectangle` peut référencer un objet de type `RoundedRectangle`.

Sous-typage, conversion de type sûre

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 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 dérivant de `A` ;
- l'objet est donc d'un type dérivant de `B`, ancêtre de `A` ;
- le type dynamique a donc tous les attributs et méthodes de `B`.

Conversion de type sûre : exemples

La conversion en un type ancêtre est souvent silencieuse et implicite, lors :

- d'une affectation dans une variable d'un classe ancêtre ;
- d'un passage en argument de classe ancêtre ;
- d'un 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();
```

Test dynamique de type : `getClass`

Découvrir le type dynamique d'une variable est possible grâce à l'introspection :

- toute classe hérite d'`Object` la méthode : `Class getClass()` ;
- `(java.lang.) Class` est une représentation réifiée d'une classe Java ;
- toute classe a un attribut statique `class` de type `Class`.

Les objets de type `Class` peuvent être alors :

- transformés en chaîne : `getName()`
- comparés : `==` (une seule instance pour chaque classe)
- inspectés pour lister les attributs, les méthodes, etc.

exemples

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

Conversion de type non sûre

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

Le type statique `A` contraint à n'accéder qu'aux méthodes et attributs de `A`.

Pour accéder aux attributs et méthodes de la classe dérivée réelle,
il est nécessaire de changer explicitement le type statique.

Opérateur de conversion : `(Classe) expr`

Si `Classe` n'est pas un ancêtre du type dynamique de `expr`,
la conversion échoue. (erreur à l'exécution : exception)

exemples

```
void f(Rectangle r) {
   RoundedRectangle g = (RoundedRectangle) r;
   r.getRadius;
}
f(new RoundedRectangle(...));    // OK
f(new Rectangle(...);            // echec dans f
```

Test dynamique de type : `instanceof`

Test de type le plus courant :
la variable de type statique `A` contient-elle en réalité
un objet d'une classe donnée, dérivée de `A` ?

Test de classe : `expr instanceof Classe`

- `expr instanceof Classe` est vrai
  si et seulement si `(Classe)expr` n'échoue pas

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

exemple

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

Application : méthode `equals` polymorphe

L'opérateur `==` effectue un test d'égalité physique 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 ;
- en réalité seul comparer des objets de même classe a un sens
  $\Longrightarrow$ utilisation d'un test dynamique de classe.

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);
    }
}
```

Sous-typage des tableaux

Typage des tableaux :

`Classe[] var` signifie :
tout élément du tableau `var` référence un objet de type `Classe` ou dérivé.

Conséquences

- si `A <: B` alors, `A[] <: B[]` (co-variance, voir cours 6)
- la conversion de `A[]` en `B[]` est implicite et acceptée :

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

- une lecture `y[0]` renvoie bien un objet dérivé de `Rectangle`
  le type statique `Rectangle` est donc correct
- une écriture `y[0] = new Rectangle()` est, par contre, **incorrecte**
  `x` contiendrait un `Rectangle`, malgré le type `RoundedRectangle[]` !

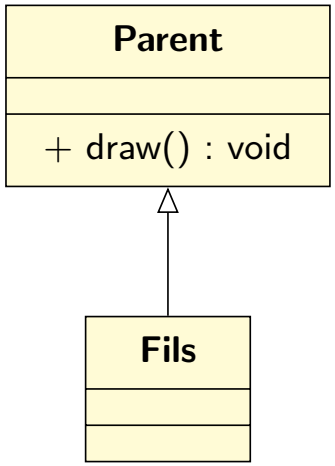
$\Longrightarrow$ la vérification statique de type est insuffisante à la sûreté de l'écriture ;
Java insère une vérification dynamique à chaque écriture !

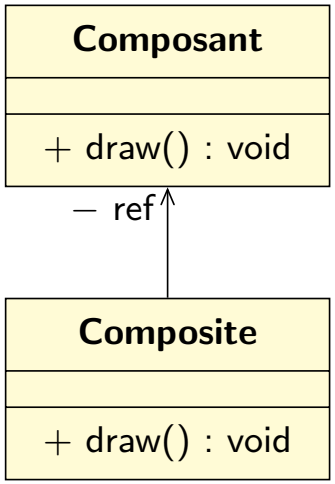
lent, et génère des erreurs de type à l'exécution au lieu de la compilation...

Héritage et composition

Héritage ou composition ?

Deux formes de réutilisation : l'héritage et la composition.





Héritage :

- mot-clé `extends`
- délégation implicite au parent
- relation "est un"
  un rectangle arrondi est un rectangle

Délégation :

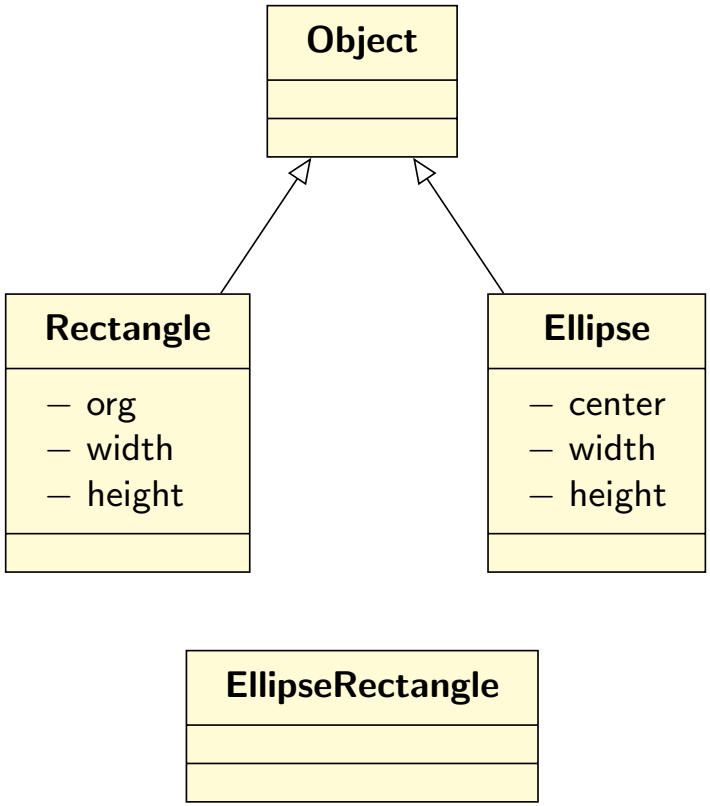
- référence à une instance
- délégation explicite à une autre classe
- relation "a un"
  un rectangle a un point

L'héritage semble plus attrayant *a priori* car il y a moins de code à écrire ;
toutefois la composition est souvent plus flexible, et plus maintenable *a posteriori*.

Règle : préférer la composition à l'héritage.

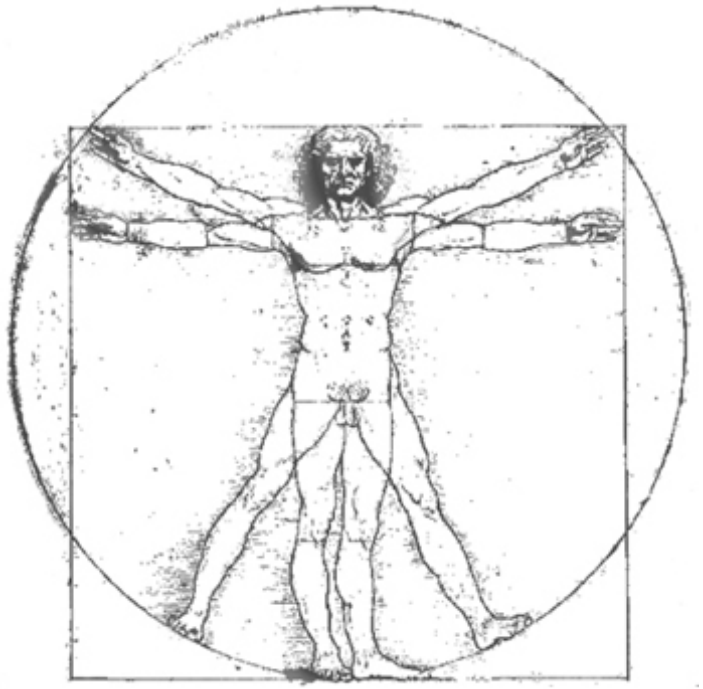
Limites de l'héritage simple

Java est limité à l'héritage simple.
Comment créer une classe qui hérite de plusieurs traits ?



Exemple :

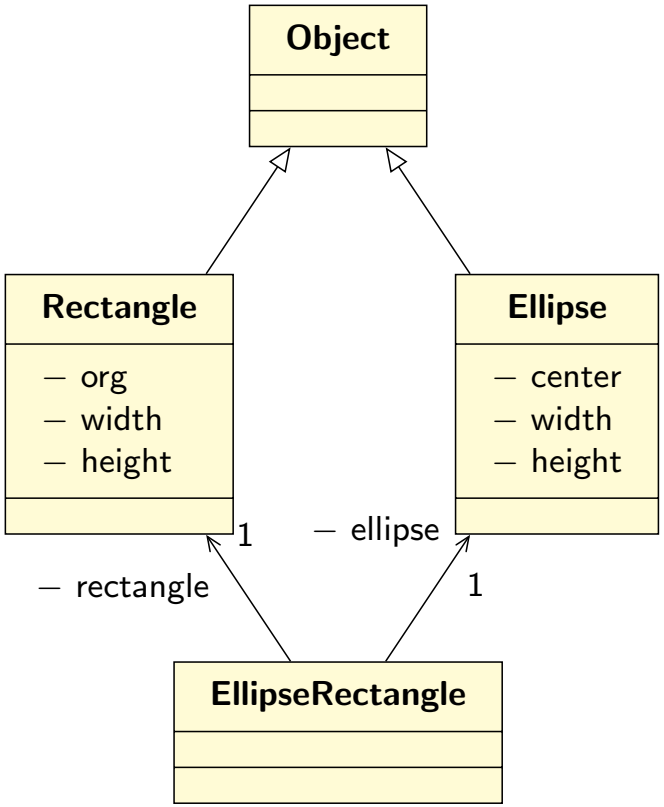
combiner une ellipse et un rectangle



Mauvaise solution :

hériter arbitrairement de `Rectangle`, et recopier le code d'`Ellipse` nécessaire...

Solution : composition



EllipseRectangle.java

```
public class EllipseRectangle {

    private Rectangle r;
    private Ellipse e;

    public EllipseRectangle(...) {
        r = new Rectangle(...);
        e = new Ellipse(...);
    }

    public void draw() {
        r.draw();
        e.draw();
    }
}
```

Autres avantages : facilite les évolutions

- ajout de nouveaux composants ;
- remplacement d'un composant par un composant dérivé, sans changer la hiérarchie de classes.

  e.g., instancier `r` avec un `RoundedRectangle`,
  inutile même de changer le type statique de l'attribut dans la déclaration

Limite de l'intuition "est un"

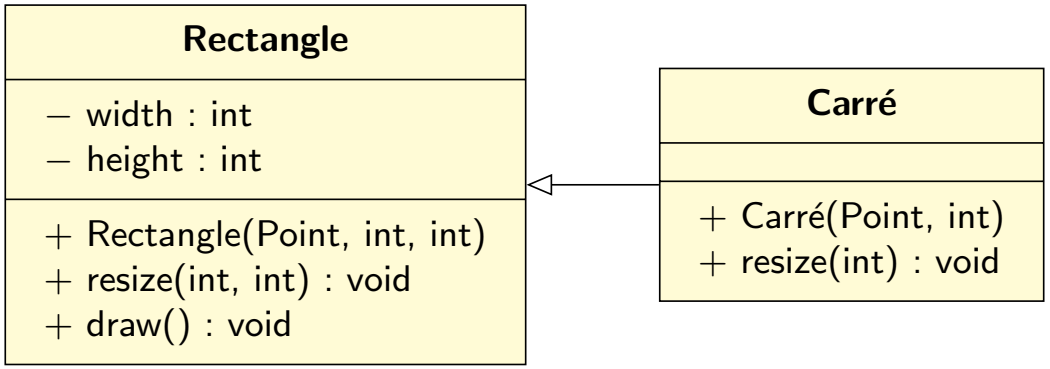
Exemple :

Étant donnée la classe `Rectangle` fixée, nous souhaitons ajouter une classe `Carré`.

Fausse solution :

- un carré est un rectangle ;
- `Rectangle` a tout ce qu'il faut pour dessiner des carrés ;

$\Longrightarrow$ dérivons `Carré` de `Rectangle` !



Carré.java

```
public Carré(Point p, int size)
{ super(p, size, size); }

public void resize(int size)
{ super.resize(size,size); }
```

Limite de l'intuition "est un" (suite)

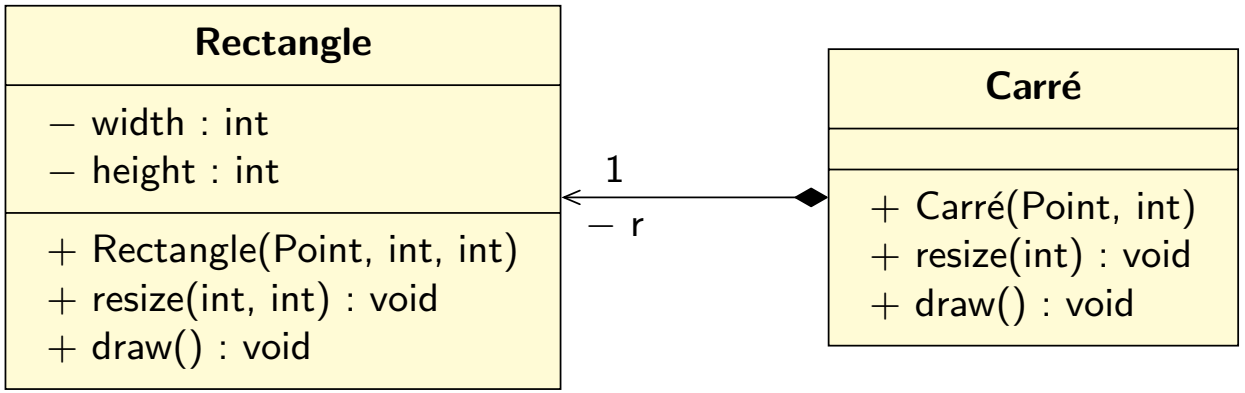
client

```
void f(Rectangle r) {
  r.set(10,20);
}

Carré c = new Carré(10);
f(c);
```

Problèmes :

- un carré est un rectangle avec ue contrainte supplémentaire : `width` = `height` ;
- par héritage, `Carré` expose une méthode `set(int,int)` qui permet à un client de briser cet invariant ;
- par sous-typage, une variable `Rectangle` peut contenir une instance de `Carré` ; le client est de bonne foi en croyant pouvoir appeler `set(int,int)` ;
- le principe de substitution de Liskov n'est pas respecté : impossible de remplacer les rectangles par des carrés et espérer que le programme continue de fonctionner !

Solution : composition



Carré.java

```
public Carré(Point p, int size)
{ r = new Rectangle(p, size, size); }

public void resize(int size)
{ r.resize(size,size); }

public void draw()
{ r.draw(); }
```

- `Rectangle` et `Carré` n'exposent plus la même interface ;
- le losange ♦ indique que la durée de vie du `Rectangle` est liée à celle du `Carré` qu'il représente.

Interfaces

Interfaces comme abstractions

La classe décrit l'implantation :
comment les objets sont représentés, crées et manipulés.

L'interface décrit une vue publique :
liste les opérations qu'un client peut faire sur un objet.

L'interface s'abstrait donc :

- de tous les attributs ; (sauf les attributs `static final`, décrivant des constantes globales)
- de tous les constructeurs ; (une interface n'est donc pas instanciable !)
- de toutes les méthodes non publiques ;
- du code des méthodes publiques ;
  seuls comptent : le nom de la méthode, le type des arguments et de retour.

À ce niveau d'abstraction
des classes différentes peuvent implanter la même interface.

Pour un client basé sur une interface, les classes l'implantant sont interchangeables
$\Longrightarrow$ le client est donc polymorphe et réutilisable.

Exemple d'une interface et ses implantations

pobj/cours2/Shape.java

```
public interface Shape {
     public void draw();
     public void translate(double x, double y);
     public void resize(int width, int height);
}
```

pobj/cours2/Rectangle.java

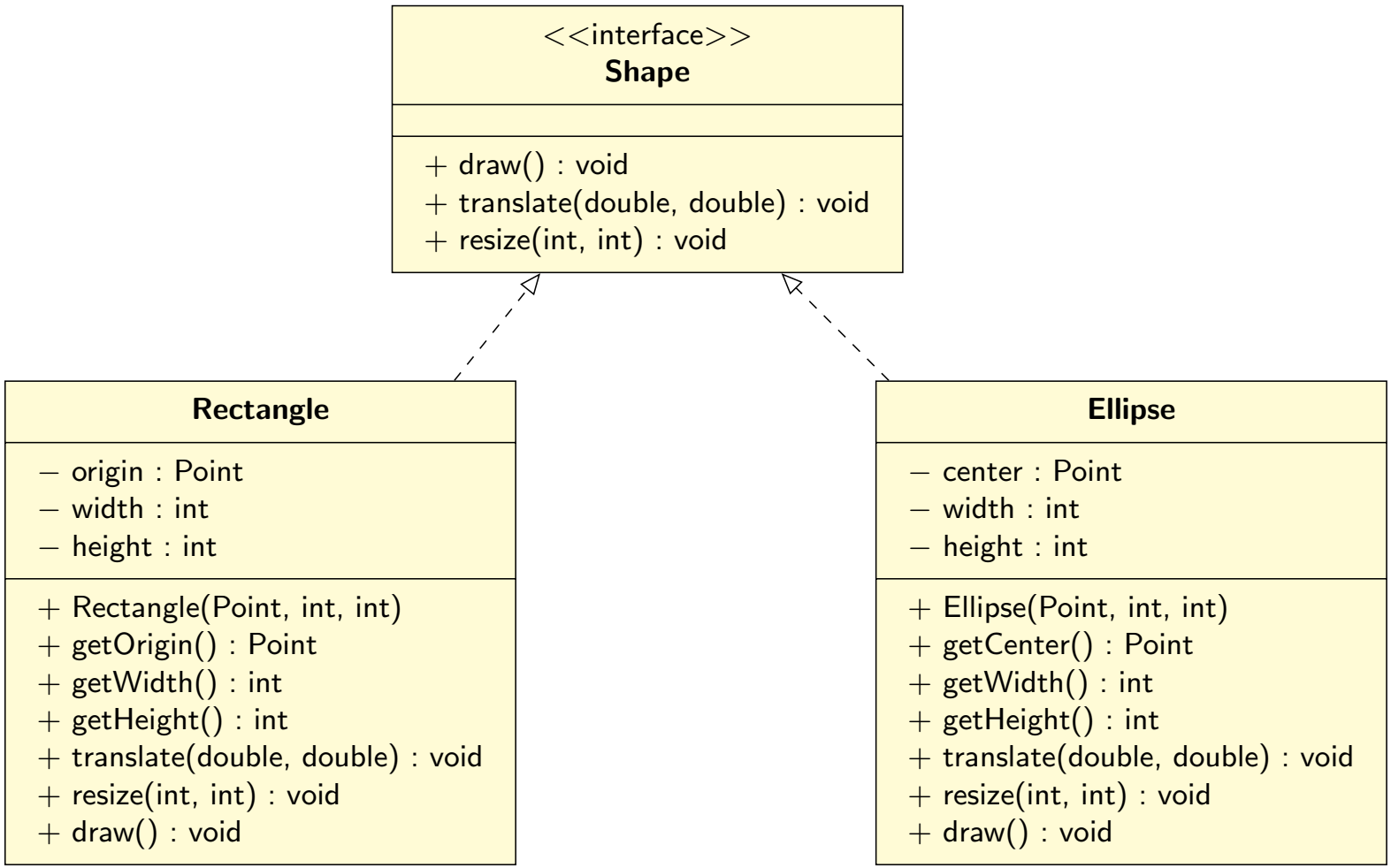
```
public class Rectangle
implements Shape {
    ...
}
```

pobj/cours2/Ellipse.java

```
public class Ellipse
implements Shape {
    ...
}
```

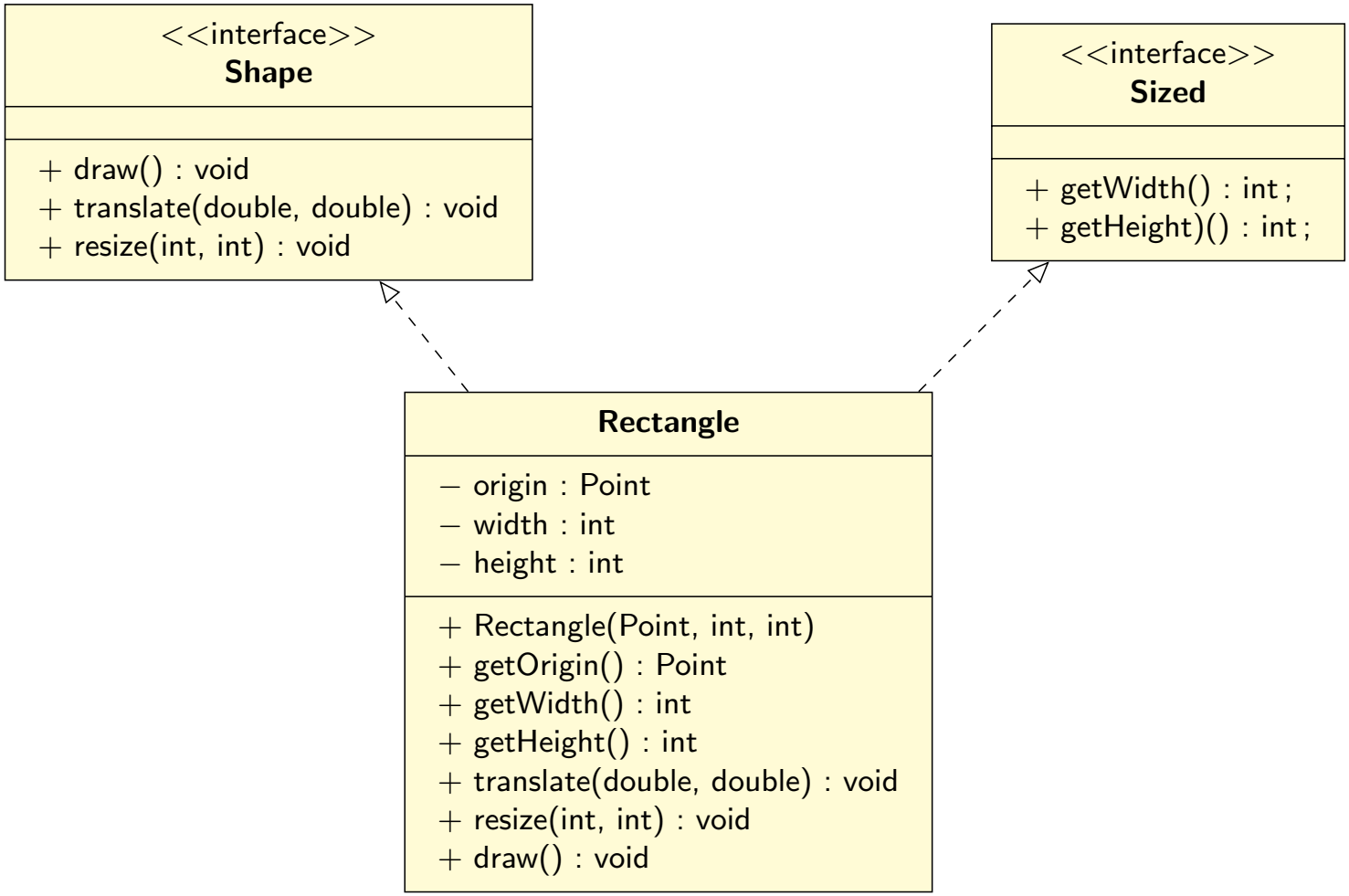
Une objet `Shape` peut être dessiné, déplacé et redimensionné, comme par exemple tous les objets `Rectangle` et `Ellipse`.

Diagramme UML d'une interface et de ses implantations



- une interface est matérialisée par le mot-clé : <<interface>> ;
- la relation "implante" est une flèche creuse en pointillés.

Implantation d'interfaces multiples



Polymorphisme du fournisseur : une classe peut implanter plusieurs interfaces :

```
public class Rectangle implements Shape, Sizeable
```

$\Longrightarrow$ encourage la réutilisabilité.

Interfaces comme types

Il est impossible de créer un objet de type dynamique instance
`new Shape(...)` est illégal (pas de constructeur).

Mais une interface est un type statique autorisé, donc on peut :

- déclarer une variable 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 peut échouer à l'exécution

Sûreté du typage :

Une variable `r` de type statique interface contiendra à l'exécution toujours une référence vers un objet obéissant à l'interface `Shape` i.e., créé par une classe implantant l'interface `Shape`.

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

Programmer avec des interfaces

Règle

Programmer vis à vis d'une interface, pas d'une implantation

Principes :

- faire le moins d'hypothèses possibles sur la représentation des objets utilisés ;
- utiliser le minimum d'attributs et de méthodes pour ses besoins ;
- être indépendant de la hiérarchie de classes.

Pour cela, un client devrait, au maximum :

- utiliser des types interfaces, et non des classes ;
- déléguer à des interfaces, et non des classes.

Exemple : `List<String> l = new ArrayList<String>();`

L'implantation `ArrayList` n'est référencée que dans la constructeur ;
nous nous forçons à n'utiliser que les méthodes de `List` dans `l` ;
$\Longrightarrow$ `ArrayList` peut être remplacée par toute implantation de `List`.

Exemple : un rectangle escamotable

dépendance sur l'implantation

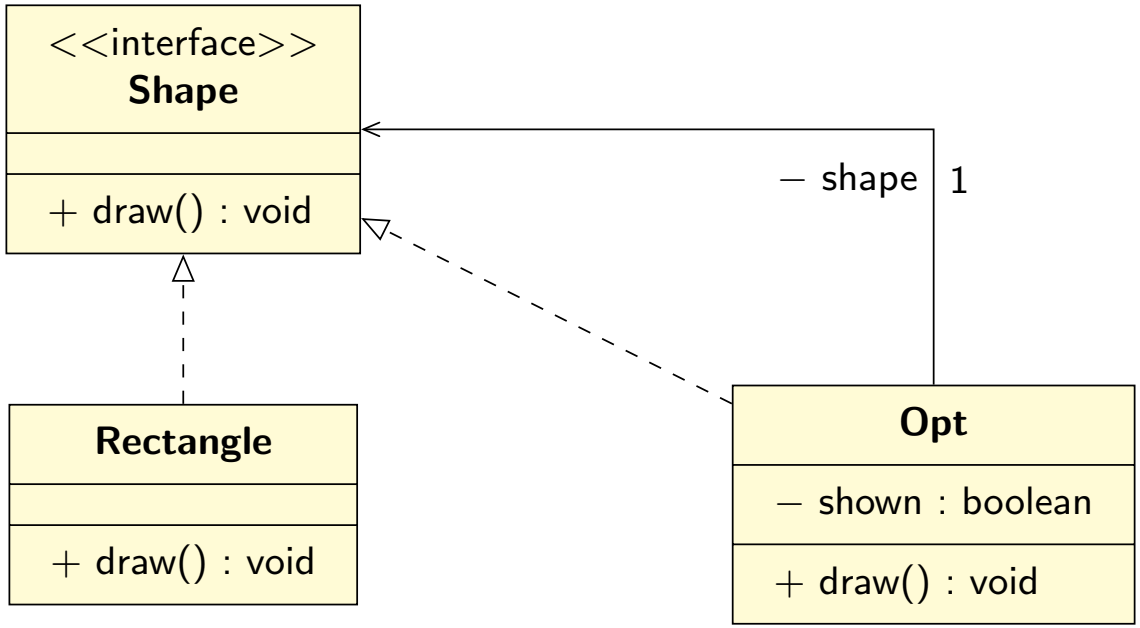
```
public class RectangleOpt
inherit Rectangle {

  private boolean show = true;

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

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

La version de gauche est plus courte
mais n'est pas généralisable à d'autres formes.

Diagramme UML, motif décorateur



`Opt` implante et délègue à la même interface `Shape`.

Ce type de constructions est très fréquent.

C'est un exemple de design pattern : le motif décorateur.

Motif : interface marqueur, `Cloneable`

`java.lang.Cloneable`

```
public interface Cloneable {
}
```

Une interface marqueur est vide.
Elle sert à indiquer une fonctionnalité non liée à une méthode.

- une classe qui fournit la fonctionalité précise `implements I` ;
- un client peut tester dynamiquement si un objet a la fonctionnalité avec `instanceof I` ;
- sert également à documenter le fournisseur.

Exemple : interface `Clonable`

indique à la JVM que la méthode `clone` peut être utilisée ;
garde-fou, car la méthode `clone` héritée de `Object` n'est pas forcément adéquate.

Note : ce motif est moins utile maintenant que langage Java comporte des annotations `@`.