Bases de Java — Héritage et interfaces

LU3IN002 : Programmation par objets

L3, Sorbonne Université

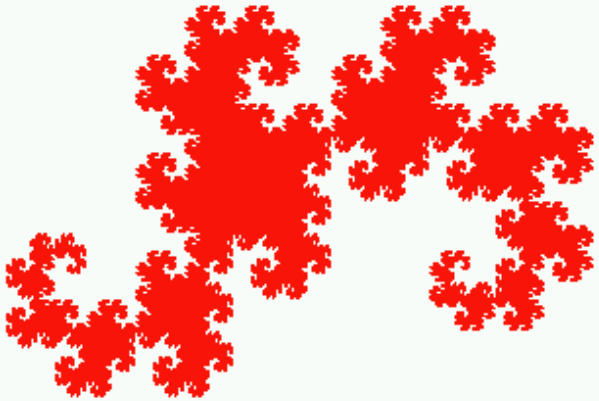
`https://moodle-sciences-23.sorbonne-universite.fr`

Antoine Miné

Cours 2

13 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 :

- héritage
- composition
- interfaces

Rappel : 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 logicielle (*design patterns*).

Principes :

1. encapsulation
   objets, classes, packages, modules
2. abstraction
   contrôle d'accès, interfaces
3. réutilisabilité
   héritage, composition
4. polymorphisme
   typage, hiérarchie de classes, liaison tardive

Fil conducteur

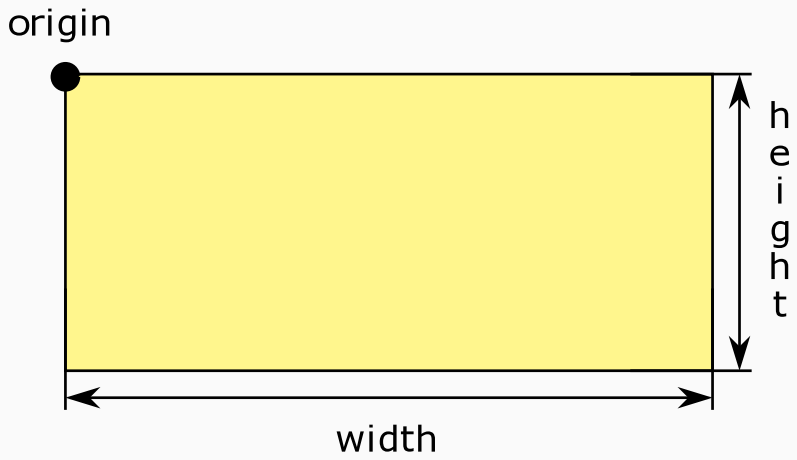
Illustration des concepts sur une application simple : un logiciel de dessin.

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

Les formes sont des objets avec :

- des attributs privés (position, taille, etc.) ;
- des invariants (la taille est positive) ;
- des constructeurs pour initialiser ces objets ;
- des méthodes publiques pour accéder à ces attributs ;
- des méthodes publiques pour modifier ces attributs ;
  (en respectant toujours les invariants)
- des méthodes privées (ou de visibilité package), utilisées en interne.

Exemple : un autre rectangle est possible



Un rectangle maintenant est défini par :

- son coin supérieur gauche : un point *origin* ;
- sa taille : des flottants *width* et *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* et *height* doivent être toujours positives.
  Cela ne peut pas être exprimé par typage en Java
  $\Longrightarrow$ ce sera donc un invariant implicite, maintenu par nos méthodes.

Rappel : la classe Java

pobj/cours2/Rectangle.java

```
package pobj.cours2;

import pobj.cours1.Point;

public class Rectangle {
  private Point org;
  private double width, height;

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

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

pobj/cours2/Rectangle.java

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

  public void resize(double w, double h) {
    if (w <= 0 || h <= 0) return;
      width = w; height = h;
  }

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

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

Nouvelle classe `Rectangle` (implantation différente de celle du cours 1).

Nous utilisons un nouveau package `pobj.cours2` pour éviter les conflits avec `Rectangle` du cours 1, tout en réutilisant la classe `Point` définie dans `pobj.cours1`.

Diagrammes de classes UML avec packages

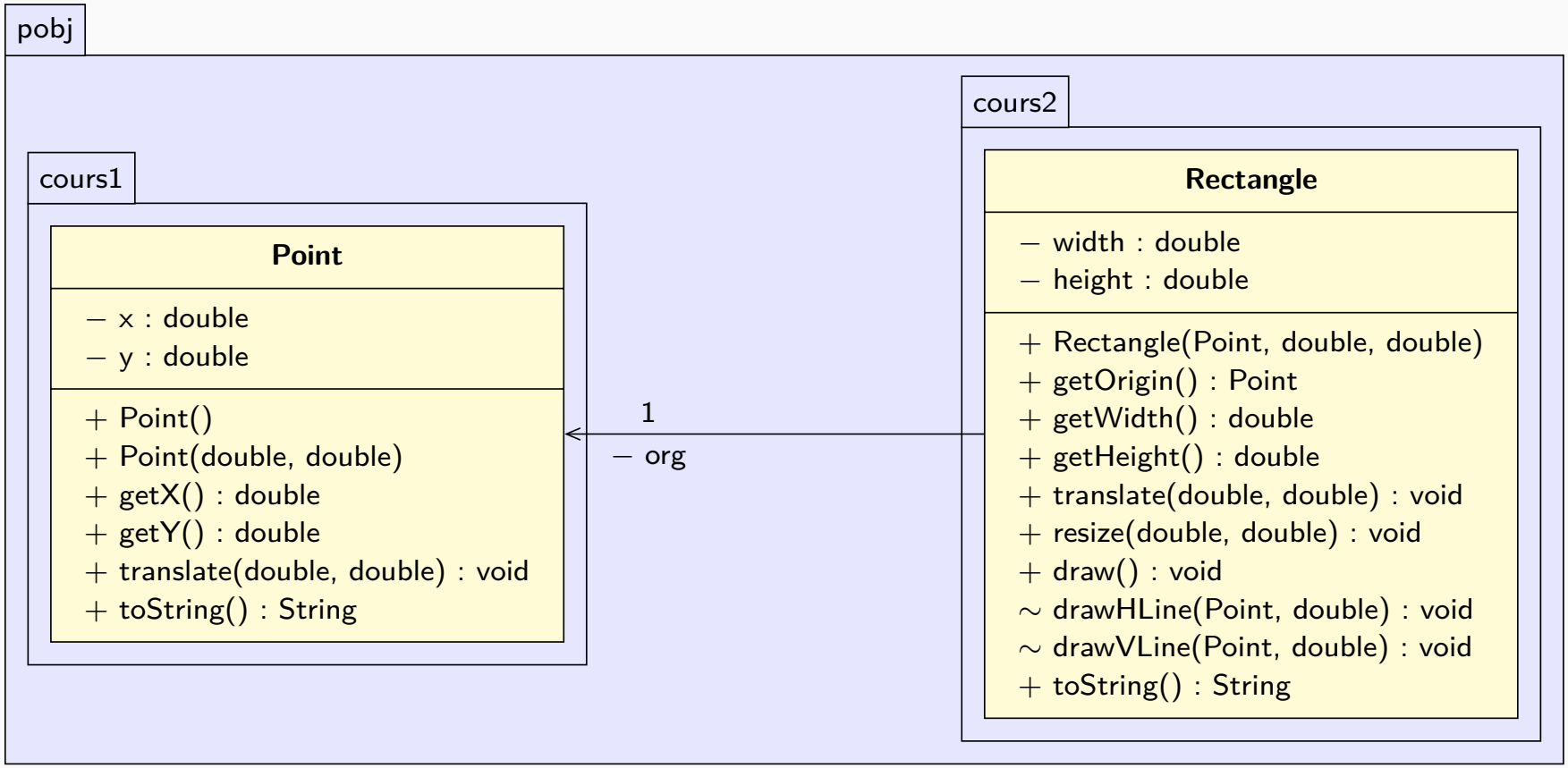


Diagramme UML montrant les classes, leur contenu et leurs relations → :

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

Note : par rapport au cours 1, la méthode `translate` de `Point` est devenu publique, pour pouvoir l'appeler depuis le package `pobj.cours2`

Le cycle de vie des objets

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

1. Le programme crée 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, retournée par `new`, est stockée, copiée, passée en argument,... par le programme.

   Le programme accède aux attributs et appelle les méthodes de l'objet, toujours 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 ne faut pas garder de référence sur un objet quand il devient inutile !
   $\Longrightarrow$ penser à mettre à `null` les attributs des objets de longue vie dès qu'ils ne sont plus utiles.

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

Rappels : valeurs et références

Java manipule plusieurs sortes des valeurs :

- des valeurs primitives :
  - entiers : `int`, `short`, etc.
  - flottants : `double`, `float`
  - caractères : `char` (16-bit unicode)
  - booléens : `boolean`
- des références sur des objets, des tableaux, ou la référence `null`

Les objets et tableaux sont toujours **manipulés par référence** :

- une affectation ou un appel de méthode copie la référence, pas l'objet
  $\Longrightarrow$ toute modification par une référence est visible par les autres références
  c'est le problème de l'aliasing ;
- l'égalité de références peut être testée avec l'opérateur `==`.

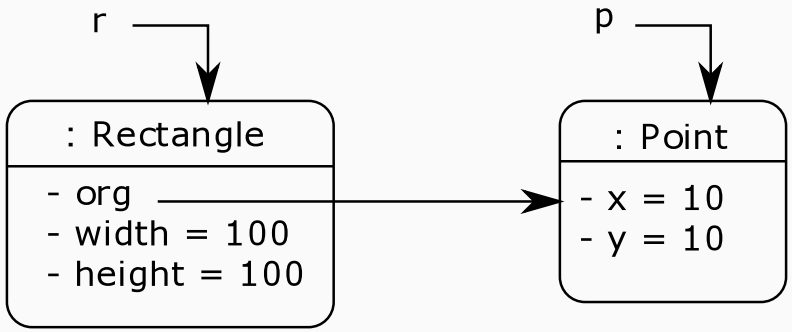
exemple

```
Point p = new Point(10, 10);
Rectangle r = new Rectangle(p, 100, 100);
p.translate(10,10);
// Le rectangle r a bougé !
// r.getOrigin() == p est vrai
```

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 objets instances existants, avec :
  - leur classe ; (rien avant le signe : car les objets n'ont pas de nom)
  - la valeur des attributs ;
- les références entre objets (par des flèches) ;
- les valeurs des variables locales (r, p).

Répertoire de compilation, classpath

rappel

```
javac pobj/cours1/Point.java pobj/cours1/Rectangle.java pobj/cours1/Programme1.java
java pobj.cours1.Programme1
```

Option `-d` de `javac` : préfixe du répertoire de compilation

- `javac /home/moi/src/pobj/cours2/Programme2.java -d` *`path`*
  crée *`path`*`/pobj/cours2/Programme2.class`
- le suffixe `pobj/cours2/Programme2.class` est déduit des directives `package` et `class` du source Java

Option `-cp` de `javac` et `java` : *classpath* : répertoires de recherche des `.class`

- `java -cp` *`path`* `pobj.cours2.Programme2`
  cherchera *`path`*`/pobj/cours2/Programme2.class`
- `javac -cp` *`path`* `x.java` cherchera aussi les dépendances `.class` de `x` dans *`path`*`/...`
- *`path`* peut être un répertoire, ou un fichier `.jar` (archive ZIP contenant une arborescence de fichiers)
- possiblité de spécifier plusieurs répertoires ou `.jar`, séparés par `:` (sous Linux)

exemple

```
javac -d bin src/pobj/cours1/Point.java
javac -d bin -cp bin src/pobj/cours2/Rectangle.java
javac -d bin -cp bin src/pobj/cours2/Programme2.java
java -cp bin pobj.cours2.Programme2
```

Packaging : JAR, modules

But : distribuer une grande quantité de classes dans une hiérarchie de packages

- fichier JAR (extension `.jar`)

  archive (ZIP) contenant une arborescence et un descripteur simple `MANIFST.MF`

pobj.jar

```
pobj/cours1/Point.class
pobj/cours1/Rectangle.class
pobj/cours2/Cercle.class
META-INF/MANIFEST.MF
```

- depuis Java 9 : les modules

  ajout d'un descripteur `module-info.java` (compilé en `module-info.class` dans le JAR)

module-info.java

```
module pobj.emu {
    requires javafx.graphics;
    export pobj.emu.main;
}
```

  - `export` : liste ce qui est fourni, avec la granularité du package
    réduit la visibilité des classes $\Longrightarrow$ meilleure encapsulation
  - `requires` : liste les modules dont ce module dépend, avec vérification à l'exécution
    $\Longrightarrow$ installation fiable d'applications complexes
  - la bibliothèque Java de base est maintenant dans le module `java.base` (toujours inclus)

Classpath et Modulepath

Le *Classpath* est composé :

- des options `-cp` passées à à `java` et `javac`
- de la variable d'environnement `CLASSPATH`
- des répertoires systèmes « en dur » du JRE.

chaque élément est un répertoire ou un JAR.

Pour profiter des modules, il faut utiliser à la place le *Modulepath* :

- option `--add-module` *`module1,module2,...`* ajoute des modules
- option `--module-path` *`path`* précise où chercher les `.jar` des modules

possibilité de mélanger classpath et modulepath, traiter des modules comme de simples JAR, ...

exemple : JavaFX

```
java --module-path /usr/lib64/openjfx-11/lib/
     --add-modules javafx.base,javafx.controls,javafx.graphics ...
```

Eclipse : Modulepath et Classpath dans « Properties > Java Build Path > Libraries ».

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`.

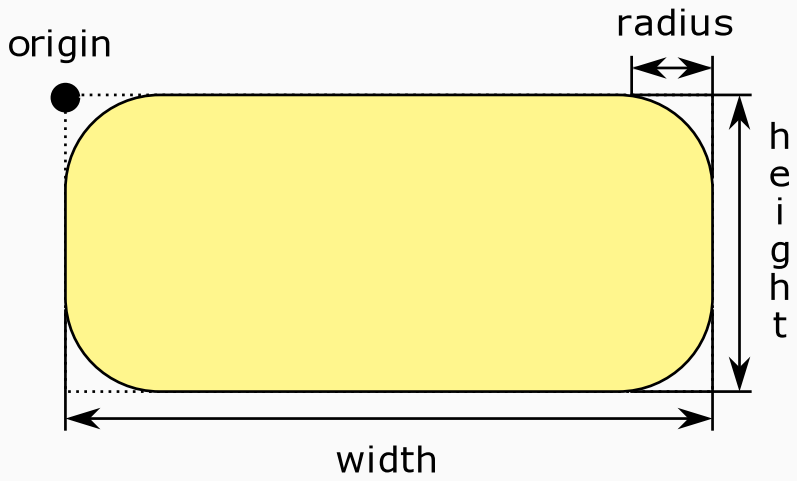
La classe dérivée peut :

- accéder aux méthodes et attributs de la classe parent ;
  sous réserve de visibilité, voir plus loin
- ajouter de nouveaux attributs et méthodes ;
- 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 : le rectangle arrondi



Un rectangle aux coins arrondis est défini :

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

Un rectangle peut être vu comme un rectangle arrondi où `radius` $= 0$

$\Longrightarrow$ le rectangle arrondi étend donc le rectangle.

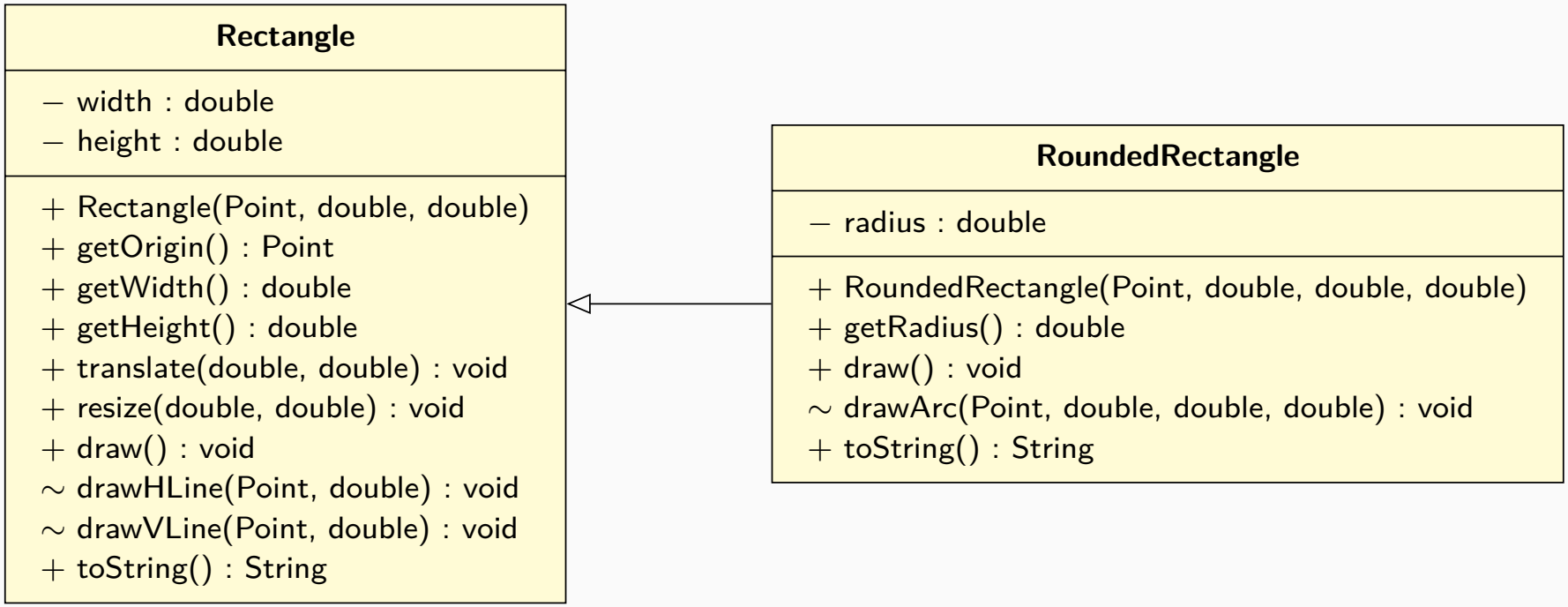
Nous voulons réutiliser au maximum le code existant de `Rectangle`.

Exemple d'héritage : le rectangle arrondi

pobj/cours2/RoundedRectangle.java

```
package pobj.cours2;
import pobj.cours1.Point;

public class RoundedRectangle extends Rectangle {

  private double radius;
  public double getRadius() { return radius; }

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

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

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

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

Héritage en UML



Notation UML de l'héritage :

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

Accès aux attributs et aux méthodes hérités

pobj/cours2/RoundedRectangle.java

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

client

```
Point p = new Point();

RoundedRectangle r =
    new RoundedRectangle(p, 10, 20, 2);

// accès de getWidth de Rectangle
double w = r.getWidth();
```

Une instance de classe dérivée possède, en mémoire,
tous les attributs d'une instance de la 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
- de stocker une instance de la classe dérivée
  dans une variable déclarée avec le type de la classe parent.
  `Rectangle r = new RoundedRectangle(...)`

Rappels : visibilité

Visibilité des attributs, méthodes et constructeurs (`new`),
du plus permissif au plus strict :

UML	visibilité	mot-clé	accès autorisé pour : classe seule	toute classe du package	toute sous-classe	toute classe
+	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 peut être de visibilité publique ou package (par défaut)
$\Longrightarrow$ *package* permet de cacher la classe en dehors du package.

Redéfinition et ajout, mot-clé `@Override`

pobj/cours2/RoundedRectangle.java

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

Une définition de méthode dans la classe héritée est :

- une **redéfinition** si elle a la même signature qu'une méthode du parent
  i.e., le même nom, et les mêmes types d'arguments ;
  par contre, le nom des arguments et le type de retour importent peu
- 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 héritée, 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 accessible dans la classe dérivée 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)
- donc utilisation similaire au mot-clé `this`

Utilité principale : extension dans la classe dérivée d'une méthode du parent
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, double width, double height, double radius)
{ super(origin, width, height); this.radius = radius; }

public RoundedRectangle(Point origin, double width, double 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 et des invariants différents.
De nouveaux constructeurs sont nécessaires pour initialiser correctement l'état des objets.

Un constructeur peut commencer par appeler, au choix :

- `super(...)` pour appeler un constructeur de la classe parent

  $\Longrightarrow$ obligatoire pour initialiser les 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

  cela a donc du sens d'avoir des constructeurs privés

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

Initialisation des attributs

Un nouvel objet a toujours 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 par Java, à une valeur par défaut (0, 0.0, `false`, `null`)

exemple

```
public class Toto {
    private int a = 2*6;  // initialisé à 12 par l'initialiseur
    private int b;        // initialisé à 14 par le constructeur
    private int c;        // initialisé à 0 par défaut
    private int d;        // initialisé à 42 par le bloc d'initialisation
    public Toto() { b = 2*7; }  // constructeur
    { d = 42; }                 // bloc d'initialisation
}
```

Note : les variables locales ne sont pas automatiquement initialisées
et c'est une erreur, signalée à la compilation, d'oublier de les initialiser !

Constructeur par défaut

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

Constructeur de classe dérivée

Un constructeur de la classe parent est toujours appelé :

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

exemple 1

```
public class A {
  private int x;
  // constructeur par défaut : x = 0
}

A a = new A(); // OK, A() existe

public class B extends A {
  public B(int x) {} // OK, super() appelé
}
```

exemple 2

```
public class A {
  private int x;
  public A(int x) { this.x = x; }
  // pas de constructeur par défaut
}
A a = new A(); // erreur, A() absent

public class B extends A {
  public B()      { super(0); }  // OK
  public B(int x) {} // erreur, A() n'existe pas
}
```

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 {
    public static final int max = 100;
    private static 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`...

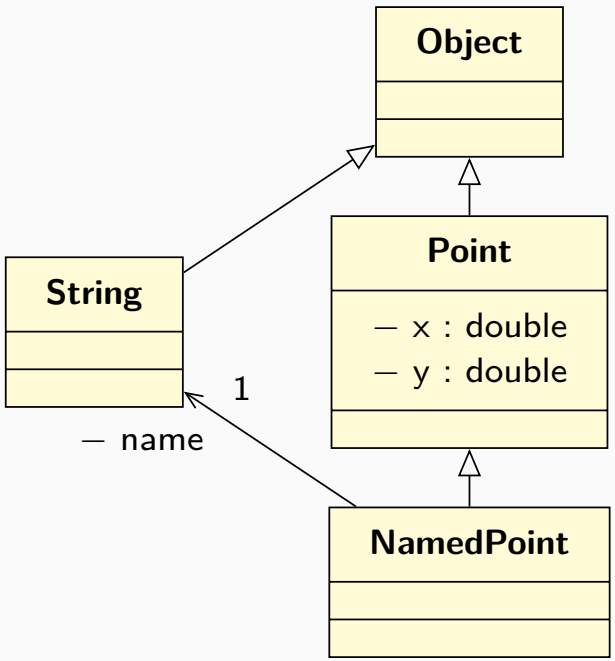
Utilisation de `static final` pour définir une constante symbolique globale.

Seul cas où un attribut `public` est justifié.

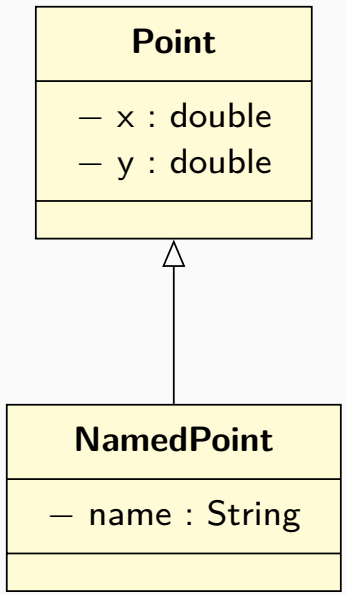
Hiérarchie de classes, classe `Object`

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

Exemple de hiérarchie : points et points nommés.



ou, plus succinctement :



Les classes bien connues, comme `Object` ou `String`, sont omises des diagrammes UML.

Rappels : méthodes standard

pobj1/cours1/Point.java

```
@Override public String toString()
{ return x + "," + y; }
```

Toute classe a des méthodes standard
avec une implantation par défaut héritée d'`Object` :

- `String toString();` — conversion en chaîne de caractères
- `boolean equals(Object obj);` — égalité (par défaut, égalité physique `==`)
- `int hashCode();` — valeur de hachage (utilisée dans les collections)
- `Class getClass();` — introspection
- `Object clone();` — copie d'objet

également : méthodes liées aux *threads* : `wait`, `notify`, `notifyAll`, ou à la gestion mémoire : `finalize`

Il est parfois souhaitable de redéfinir le comportement de ces méthodes :

- `toString` par défaut est peu informatif
- `equals` par défaut est équivalent à `==`
- `clone` (transparent suivant)

Copie d'objets, méthode `clone`

L'affectation (=) copie une référence sur un objet, sans copier l'objet.

Il est parfois nécessaire de créer un nouvel objet :
une copie de l'objet manipulable indépendemment 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 :

- avoir le choix entre :
  - une copie de surface, qui référence les attributs de l'original ;
  - ou une copie profonde, qui copie récursivement les attributs ;
- spécifier un type de retour plus précis que `Object`.

Point.java

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

Rectangle.java

```
public class Rectangle {
  private Point org;
  private double width, height;
  public Rectangle(...) ...
  @Override public Rectangle clone() {
    // au choix : copie profonde
    return new Rectangle(org.clone(), width, height);
    // ou : copie de surface
    return new Rectangle(org, width, height);
  }
}
```

Conversion de classe

- Conversion vers une classe parent :

```
Rectangle r;
r = new RoundedRectangle(...);
```

  - implicite : `RoundedRectangle` → `Rectangle`
  - toujours possible (qui peut le plus peut le moins)

- Conversion vers une classe dérivée :

```
Object o = new Rectangle(...);
Rectangle x = (Rectangle) o;
```

  - nécessite une conversion explicite : `(c) o` → conversion en classe `c`
  - nécessaire pour stocker dans une variable de type `c`, pour accéder aux attributs et méthodes de `c`
  - échoue si l'objet n'est pas de classe `c` (ou dérivée)

- Test de convertibilité : `instanceof`
  - `o instanceof C` : `o` est-il de classe `C` ? (ou dérivée)
  - `o instanceof Rectangle` → vrai
  - `o instanceof RoundedRectangle` → faux

- Test et conversion combinés : (Java 14)

```
if (o instanceof Rectangle x) { x....}
```

(plus de détails dans le cours sur le typage)

Record en Java 14

Java 14 introduit la définition de types record (enregistrements immuables) :

Point.java

```
public record Point(int x, int y) { }
```

Génère une classe `Point` immuable, avec :

- des attributs : `private final int x,y;`
- un constructeur : `public Point(int x, int y)`
- des getters : `public int x()` (pas de préfixe get)
- `public String toString()` qui liste la valeur des attributs
- `public boolean equals(Object o)` qui vérifie que o est un `Point` dont les attributs sont égaux à ceux de `this` (par `equals`)
- `public int hashCode()` qui appelle `hashCode` sur les attributs
- la classe est `final` (pas d'héritage possible)

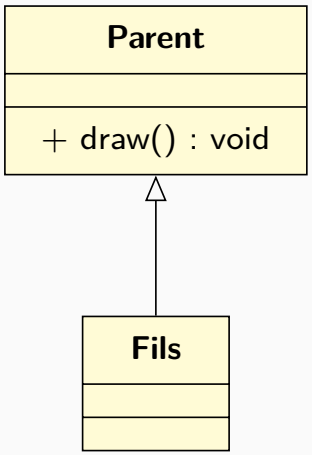
C'est un raccourci utile !

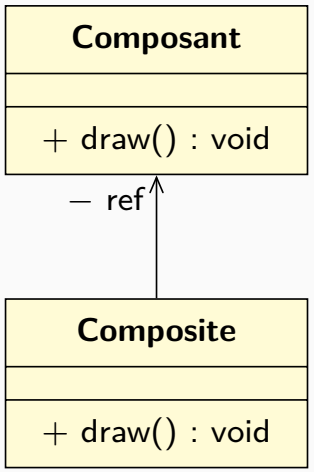
(possibilité d'ajouter des constructeurs « maison », des attributs et méthodes statiques)

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

Composition :

- référence à une instance `ref`
- délégation explicite à une autre classe
  `draw()` appelle `ref.draw()`
- relation « a un »
  un rectangle a un point

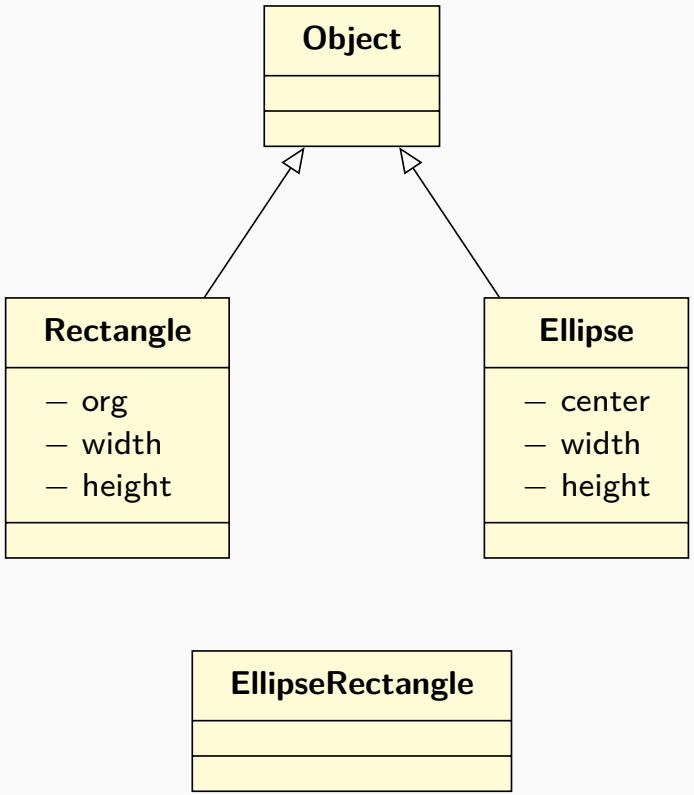
L'héritage semble plus attrayant *a priori* car il y a moins de code à écrire ; toutefois la composition est souvent plus flexible et plus facile à maintenir.

Règle : préférer la composition à l'héritage.

Limites de l'héritage simple

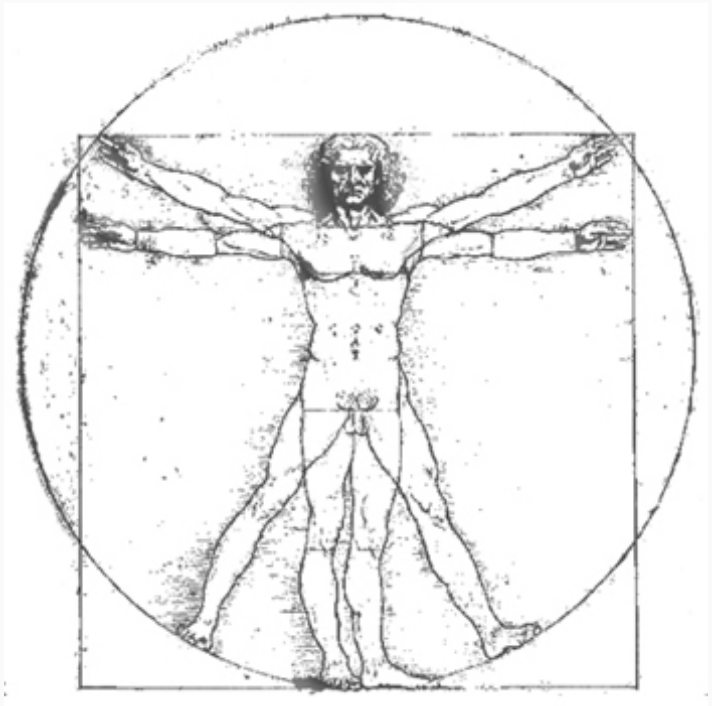
Java est limité à l'héritage simple.

Comment créer une classe qui hérite de plusieurs traits ?



Exemple :

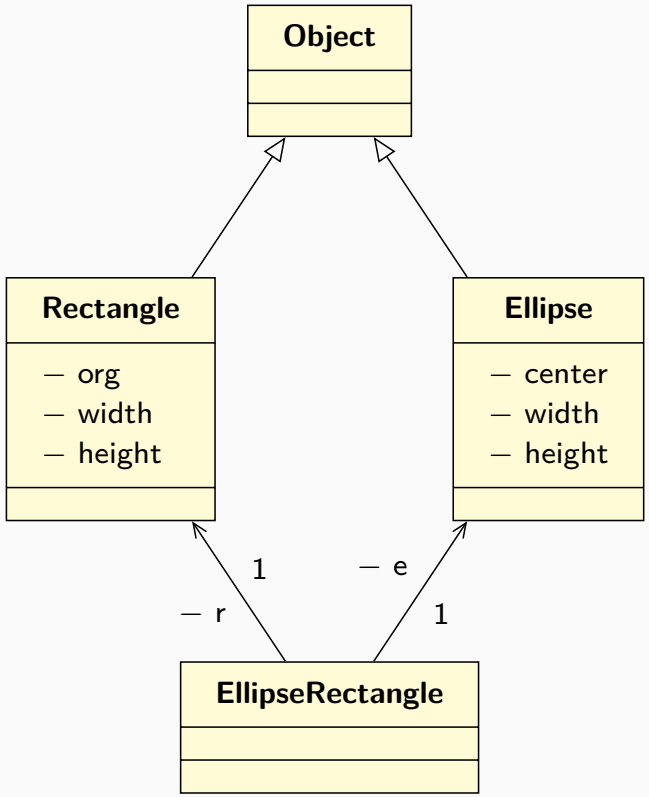
combiner une ellipse et un rectangle



Mauvaise solution :

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

Solution : composition



Composition de `Ellipse` et `Rectangle`

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();
    }
}
```

Avantages de la composition :

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

  e.g., instancier `r` avec un `RoundedRectangle` dans le constructeur : `r = new RoundedRectangle(...)`
  inutile de changer le type `Rectangle` dans la déclaration de l'attribut `r`

Limite de l'intuition « est un »

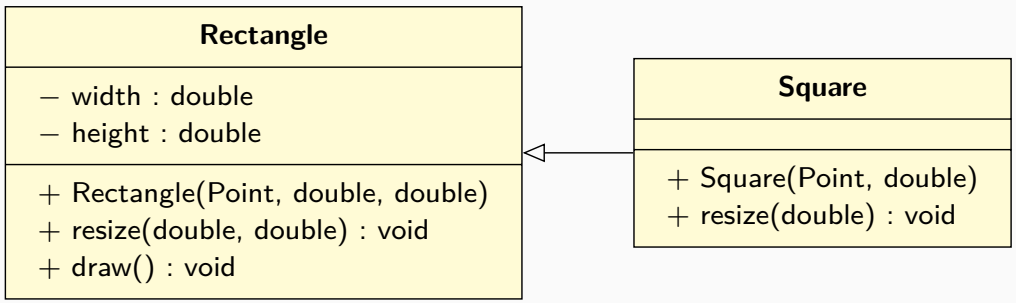
Exemple :

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

Fausse solution :

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

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



Square.java

```
public class Square extends Rectangle
{
  public Square(Point p, double size)
  { super(p, size, size); }

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

Limite de l'intuition « est un » (suite)

client

```
void f(Rectangle r) {
  // r peut référencer une instance de Rectangle ou de Square !
  r.resize(10,20);
}

Square c = new Square(10);
f(c); // conversion implicite de référence de Square en Rectangle
```

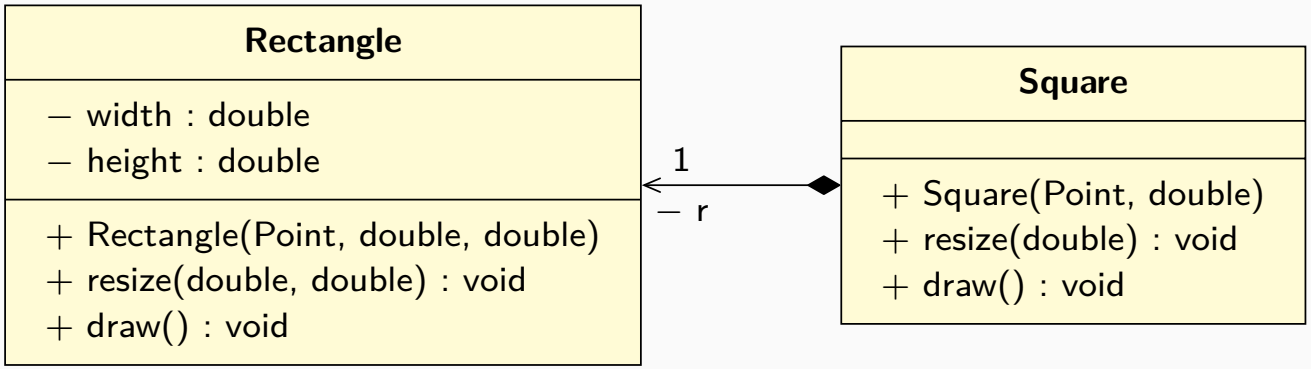
Principe de l'héritage :

- une instance de `Square` a tous les attributs et méthodes d'un `Rectangle`
- donc une instance de `Square` peut être fournie là où une instance de `Rectangle` est demandée (conversion de référence implicite lors d'affectations et passages d'arguments)

Problème :

- un carré est un rectangle avec une contrainte supplémentaire : `width` $==$ `height` ;
- or, par héritage, Square expose une méthode `resize(double,double)` qui permet à un client de briser cet invariant ;
- une variable de type `Rectangle` peut en réalité contenir une instance de `Square` $\Longrightarrow$ le client qui attend de bonne foi un `Rectangle` ne fonctionnera plus ;
- le principe de substitution « puisqu'un carré est un rectangle, on peut utiliser un carré là où un rectangle est attendu » n'est pas respecté.

Solution : composition



Square.java

```
class Square {
  private Rectangle r;

  public Square(Point p, double size) { r = new Rectangle(p, size, size); }

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

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

Solution : composition et délégation explicite

- `Rectangle` et `Square` n'exposent plus les mêmes méthodes ;
  `Square` n'a pas de `resize(double,double)` et ne peut pas être pris pour un `Rectangle`
- le losange ♦ indique en UML que la durée de vie du `Rectangle` est liée à celle du `Square` qu'il représente.

Interfaces

Interfaces comme abstractions

La classe décrit une implantation :
comment les objets sont représentés, créés et manipulés.

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

Java possède une notion d'interface au niveau du langage.

L'interface s'abstrait :

- de tous les attributs (sauf les attributs `static final`, décrivant des constantes globales)
- de tous les constructeurs (une interface n'est 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, sans relation d'héritage, peuvent implanter la même interface.

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

Mots-clés interface et implements

pobj/cours2/Shape.java

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

pobj/cours2/Rectangle.java

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

pobj/cours2/Ellipse.java

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

Une classe implante une interface si elle définit au moins les méthodes demandées avec une signature compatible.

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

Le mot-clé `public` est optionnel, les méthodes des signatures sont publiques par défaut.

Diagramme UML d'une interface et de ses implantations

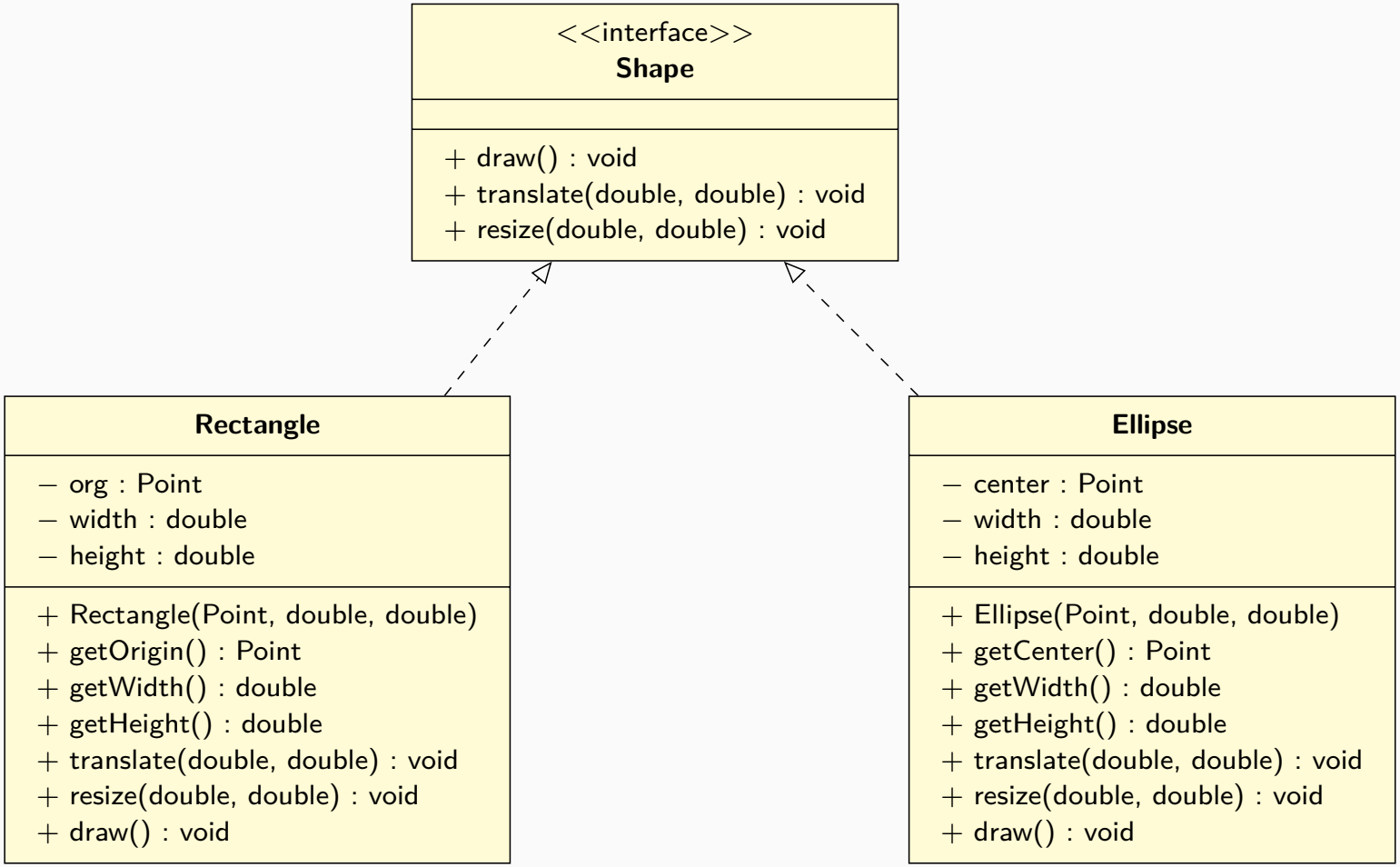


Diagramme UML :

- une interface est matérialisée par le mot-clé <<interface>> entre chevrons ;
- la relation « implante » est une flèche creuse en pointillés $--\triangleright$.

Exemple d'interface : introduction aux listes

La bibliothèque standard Java contient des structures de données très utiles comme les listes (un exemple de collection) :

- interface : `java.util.List`
- implantations : `java.util.ArrayList`, `java.util.LinkedList`, ...
  même jeu d'opérations, mais des complexités algorithmiques différentes
- `List<E>` : type des listes d'éléments de `E`
  utilisation de génériques : polymorphisme paramétrique, étudié dans un prochain cours

Quelques opérations : (voir la documentation de l'API Java pour plus d'information)

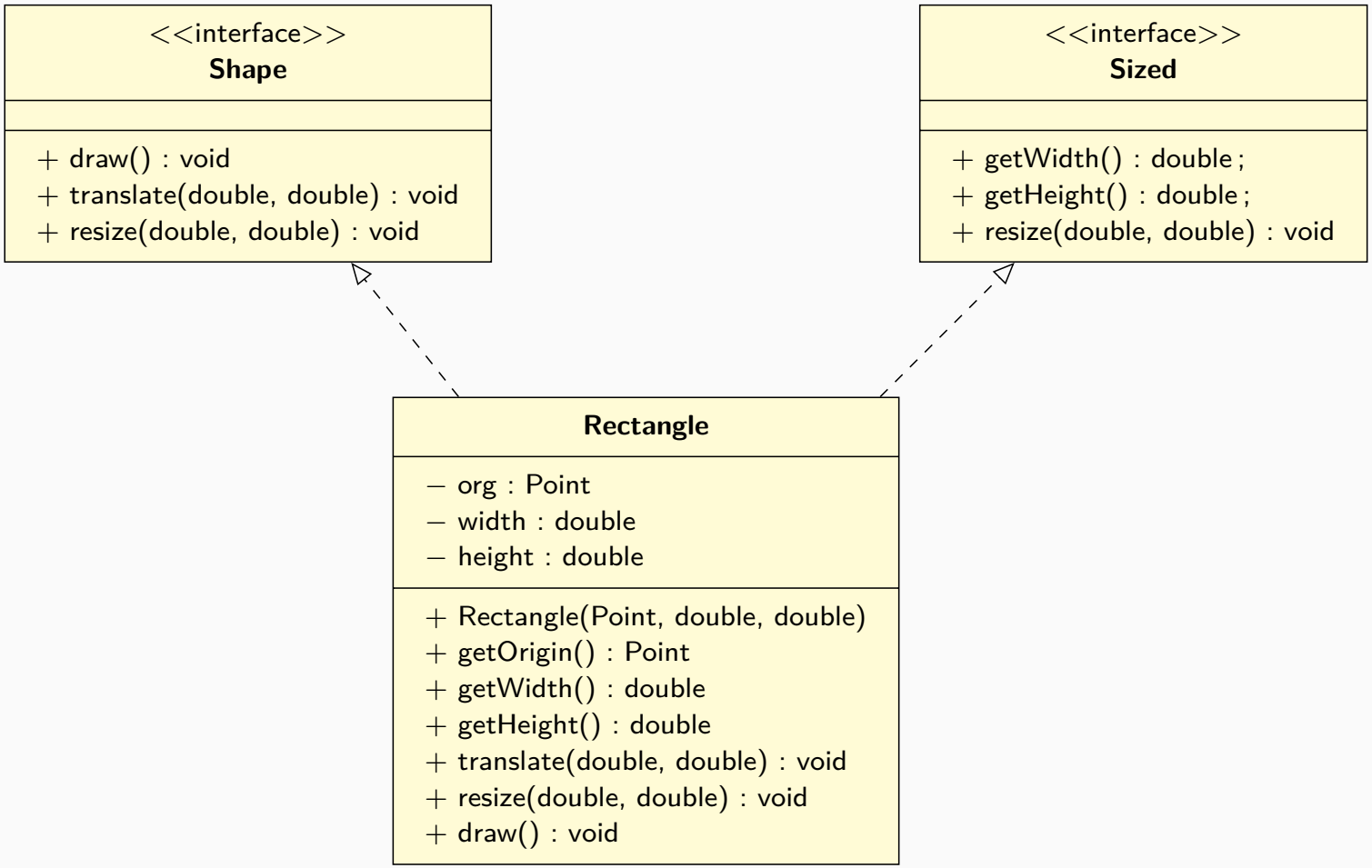
- ajout : `boolean add(E)`
- taille : `int size()`
- accès : `E get(int)`
- vide : `void clear()`
- itération : forme spéciale de `for`
  `for (type var : expr) inst`

exemple

```
List<Point> x = new ArrayList<Point>();
x.add(new Point(12,10));
x.add(new Point();
x.get(0);
for (Point p : x) p.translate(10,10);
```

Note : il est possible de déclarer une variable de type interface
c'est même conseillé pour limiter la dépendance à l'implantation !
(plus sur les listes, les collections, les types génériques dans les prochains cours)

Implantation d'interfaces multiples



Polymorphisme du fournisseur : une classe peut implanter plusieurs interfaces

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

$\Longrightarrow$ encourage la réutilisabilité.

Interfaces et héritage

Héritage d'implantation :

Si une classe `A` implante l'interface `I`
et `B` hérite de `A`,
alors `B` implante automatiquement l'interface `I`.

Inutile de préciser : `class B implements I`.

Héritage entre interfaces :

Une interface peut hériter d'une
ou de plusieurs interfaces :
`interface A extends B, C, D`.

Exemple :

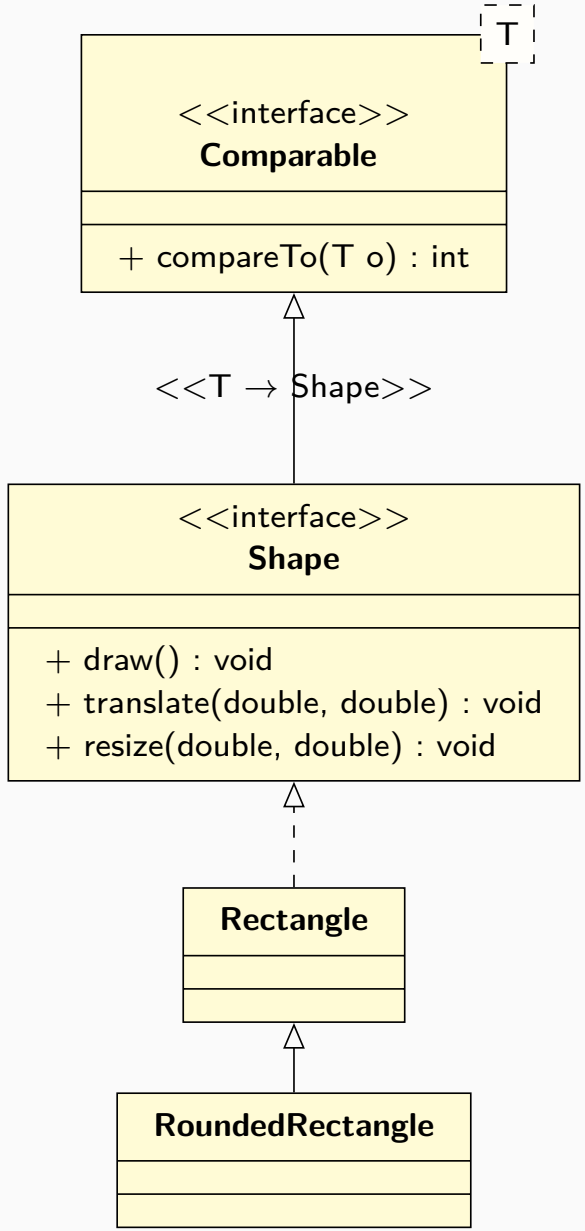
`Comparable<T>` est une interface standard définissant une méthode `int compareTo(T)`, utile pour le tri de collections.

`T` est un paramètre de type.

`interface Shape implements Comparable<Shape>`

indique que `Shape` spécialise `compareTo` avec la signature :

`int compareTo(Shape s)`



Programmer avec des interfaces

Règle

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

On ne peut pas instancier un type interface
mais on peut déclarer des variables avec un type interface
et y stocker des instances de classes implantant cette interface.

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

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

But :

- 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, le plus possible :

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