Introduction — Les bases du langage Java

LU3IN002 : Programmation par objets

L3, Sorbonne Université

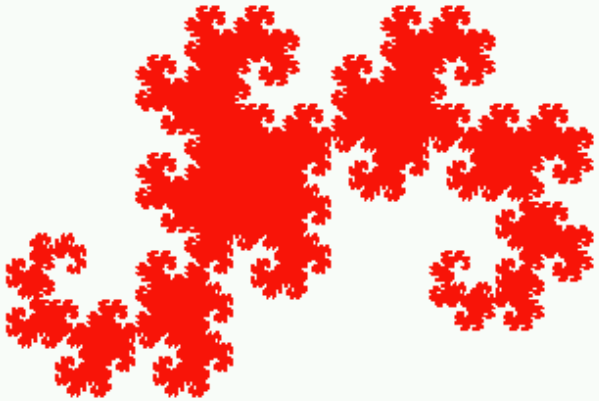
`https://moodle-sciences-23.sorbonne-universite.fr`

Antoine Miné

Cours 1

6 septembre 2023

Année 2023–2024



SORBONNE UNIVERSITÉ

LIP6

Objectifs du cours

Objectifs :

1. comprendre les concepts de la programmation orientée objet (POO)
2. maîtriser le langage Java
   - réalisation de la POO en Java
     d'autres langages OO existent !
   - typage
   - découverte de traits non spécifiques à la POO
     génériques, *lambda expressions*, flux (ajouts récents)
3. s'initier aux pratiques de la conception orientée objet
   - *Design patterns*
   - représentation UML

Pour cela nous allons :

- écrire du code Java pour réaliser des petites applications
  en TME
- concevoir des morceaux d'application (sur le papier)
  en TD

Organisation de l'UE

Chargés de cours :

- Antoine Miné
- Jonathan Lejeune

Chargés de groupe de TD et TME :

1. Grégoire Bussone (lundi)
2. Antoine Miné (mardi)
3. Yann Thierry-Mieg (vendredi)

Respectez le groupe où vous êtes inscrit !

Livret de TD imprimés et disponibles par l'ALIAS,
également disponibles en ligne sur Moodle (ne pas imprimer !)

Supports de TME uniquement en ligne sur Moodle (inutile de les imprimer).

Transparents des cours mis en ligne sur Moodle.

Environnement logiciel

Déjà disponible à la PPTI, facile à installer sur votre ordinateur

logiciels libres, disponibles sur Linux, MacOS X, Windows

- Java version 17 (ou supérieure)

  dernière version LTS
  OpenJDK disponible sur `https://jdk.java.net`, ou sous Ubuntu et Debian avec `apt`

- Eclipse IDE (environnement de développement)

  `http://www.eclipse.org/downloads`

  pour éditer, compiler et exécuter du Java, mais aussi : débugger et tester

- La ligne de commande

  savoir compiler et exécuter du Java en ligne de commande, même si on l'utilise peu

- GitLab `https://stl.algo-prog.info` (instance privée pour le cours)

  travail en binôme, dépôts de sources avec gestion de versions `git`
  intégration continue (compilation et tests automatiques à chaque `push`)

Évaluation

- partiel : 40 %

  épreuve sur machine avec correction automatique, 2h

- examen final : 45 %

  épreuve sur papier, 2h

- contrôle continu : 15 %

  rendu hebdomadaire de TME

Nombreuses annales (en grande partie corrigées) disponibles sur Moodle

TME

Mini-projets sur une ou plusieurs séances de TME :

- TME 1–3 : création de grilles de mots-croisés
- TME 4–5 : structure de multi-ensembles
- TME 6 : entraînement au partiel sur machine (annale)
- TME 7–8 : arbres d'expression et calcul symbolique
- TME 9–11 : logiciel de dessin (JavaFX)

Méthode de travail :

- travail (et rendu) par binôme
- début en séance de TME, premier rendu obligatoire
- finalisé à la maison, rendu final avant la séance de TME suivante
- utilisation de git : *fork* du projet maître, rendu par *push* et *release*
- sur un GitLab privé : `https://stl.algo-prog.info`
  inscription automatique : vous serez contactés par email pour activer votre compte
- auto-évaluation par test unitaire et intégration continue

$\Longrightarrow$ s'initier aux bonnes pratiques du développement logiciel.

Plan du cours

- Cours 1 : Bases 1 : rappels de Java et de programmation orientée objet
- Cours 2 : Bases 2 : héritage, composition, interfaces
- Cours 3 : Bases 3 : typage, liaison dynamique
- Cours 4 : Collections, itérateurs
- Cours 5 : Programmation robuste, exceptions, tests unitaires
- Cours 6 : Design patterns I : Design Patterns structurels
- Cours 7 : Polymorphisme et génériques
- 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 :

- généralités sur la POO et sur Java
- bases du langage Java

Généralités

La programmation orientée objet

But : programmer de manière

- robuste
- extensible

Concepts de base :

1. encapsulation
2. abstraction
3. réutilisation
4. polymorphisme

Ces concepts sont mis en œuvre grâce :

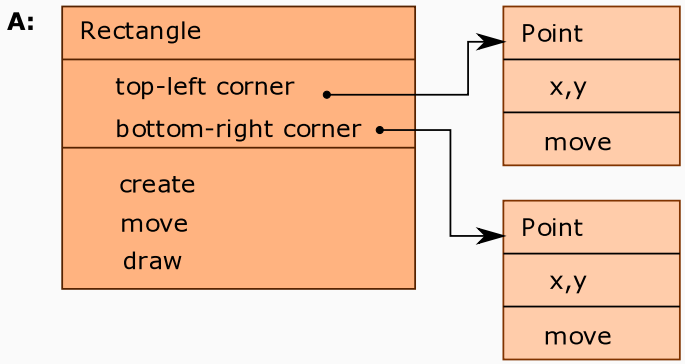
- au langage Java
- aux bonnes pratiques de programmation
- aux *design patterns* : briques réutilisables en conception logicielle

Concept objet 1 : Encapsulation

Encapsulation :

Un **objet** regroupe dans une même unité :

- un ensemble de données : les **attributs**
- le code permettant de les manipuler de manière cohérente : les **méthodes**



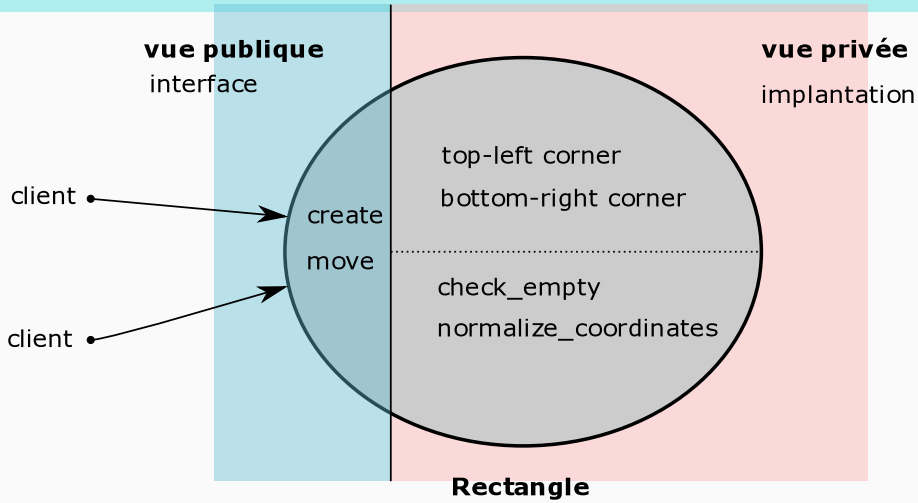
un rectangle a deux coins, un coin est un point à deux coordonnées ;
les rectangles et les points peuvent être bougés

Mécanismes Java :

- les **classes** décrivent des objets ayant la même liste d'attributs (la valeur varie généralement d'un objet à l'autre) et les mêmes méthodes
- et, à une granularité plus élevé, les packages (organisation hiérarchique) et les modules

$\Longrightarrow$ Pas de variable globale ! Tout l'état est encapsulé dans des objets (états locaux)

Concept objet 2 : Abstraction



Abstraction : distinguer

- les choix d'implantation, privés
- l'interface vers le client, publique

Bénéfices :

- se concentrer sur les fonctionnalités des objets, pas les choix d'implantation
- compréhension et utilisation facilitées des objets, en ignorant les détails
- protection : éviter la corruption accidentelle de l'état des objets
  e.g., maintenir l'invariant : le coin haut-gauche est plus en haut que le coin bas-droite
- robustesse aux changements d'implantation
  e.g., redéfinir les rectangles à l'aide d'un seul coin et d'une taille

Mécanismes Java :

- le contrôle d'accès aux attributs et méthodes
  $\Longrightarrow$ les attributs sont presque toujours privés !
- les interfaces : vue(s) publique(s) des objets
  $\Longrightarrow$ programmer vis à vis d'une interface, pas d'une implantation

Concept objet 3 : Réutilisation

Réutilisation :

- programmer une fois, utiliser plusieurs fois
- par agrégation : combiner des objets existants
  un rectangle est défini par deux points
- par délégation : déléguer le travail à des objets existants
  un dessin contient des rectangles
  afficher le dessin se réduit à afficher ses rectangles
- par spécialisation, en ne redéfinissant que ce qui a changé
  un carré est un cas particulier de rectangle
  le code du rectangle peut être utilisé pour le carré

Mécanismes Java :

- les classes : tous les objets d'une même classe ont les mêmes méthodes
- les références : déléguer à un ou plusieurs autres objets
- l'héritage : redéfinir une partie des comportements, réutiliser le reste

Java n'offre que l'héritage simple (pour les implantations) ;
la délégation est souvent plus flexible que l'héritage !

Concept objet 4 : Polymorphisme

Polymorphisme :

- un même objet peut être utilisé dans plusieurs contextes
  un carré peut être utilisé partout où un rectangle est attendu
- collections : le comportement d'un agrégat dépend peu des objets agrégés
  pour manipuler une liste de rectangles, inutile de savoir s'il s'agit de rectangles ou de carrés
- le comportement d'un objet est paramétré par un autre objet
  construire une liste ordonnée de rectangles nécessite de savoir ordonner les rectangles

$\Longrightarrow$ facilite la réutilisation

Mécanismes Java :

- implantation d'interfaces multiples
  plusieurs vues publiques d'un même objet
- l'héritage avec liaison tardive
- la surcharge de méthodes
  méthodes de même nom mais d'arguments de type différents
- les types génériques
  polymorphisme paramétrique, en gardant la sûreté du typage

Le langage Java : origine et but

Quelques langages à objets : vision historique

- Simula 67 : extension objet d'ALGOL 60
- Smalltalk (début 1980s)
- C++ (1983) extension objet de C
- Modula 3 (milieu 1980s)
- Python (pré-version en 1991)
- Java (1996 pour Java 1.0)
- C# (2000)

Java :

- créé dès 1991 par James Gosling chez Sun Microsystems
- langage moderne, sans attache, mais avec de nombreuses influences
- *Write Once, Run Anywhere*
- langage sécurisé pour le Web (intégration aux serveurs et aux navigateurs)

Le langage Java : caractéristiques

- orienté objet avec un système de classes
  mais héritage simple, contrairement à C++
- syntaxe inspirée par le C
- typage statique et fort (les variables doivent être déclarées, avec leur type)
  $\Longrightarrow$ garantie statique de sûreté dès la compilation
- gestion automatique de la mémoire (*garbage collector*)
- des exceptions
  $\Longrightarrow$ garantie dynamique de sûreté, à l'exécution
- du polymorphisme
  polymorphisme d'objet, polymorphisme de surcharge, polymorphisme paramétrique
- de l'introspection
  $\Longrightarrow$ permet la métaprogrammation (IDE, débogueur, ...)
- compilation vers du code-octet (*byte-code*)
  $\Longrightarrow$ portabilité, même après compilation
- exécution dans une machine virtuelle
  - vérification des classes au chargement
  - chargement dynamique de classes (disque, réseau)
- une bibliothèque standard très riche
  structures de données, clients et serveurs internet, *threads*, interfaces graphiques, etc.

Le langage Java : évolution

Versions du langage :

- Java 1.0, 1996
- Java 1.1, 1997 : classes internes, réflexion
- Java 1.2, 1998 : collections, JIT
- Java 1.3, 2000
- Java 1.4, 2002 : assertions
- Java 5, 2004 : changement de numérotation, types génériques, autoboxing, énumerations, boucles *for each*
- Java 6, 2006 : annotations
- Java 7, 2011 : reprise par Oracle, améliorations mineures aux `switch` et exceptions, inférence de type
- Java 8, 2014 : JavaFX, aspects fonctionnels : lambdas, flux,
- Java 9, 2017 : modules
- Java 10, 2018 : inférence de type pour les variables locales
- . . . évolution au rythme d'une version tous les 6 mois, LTS tous les 2–3 ans
- Java 16, 2021 : classes scellées, record
- Java 17, 2021 : LTS (long-term support)

Enrichissement de la bibliothèque standard : de 212 classes pourJava 1.0 à 4411 pour Java 11

Les bases du langage Java

Expressions

Les expressions Java sont basées sur celles du C :

constantes	`2`, `1.2`, `true`, `'a'`, `"toto"`
variables	`x`
opérations unaires	`- x`
opérations binaires	`2 + 2`
parenthèses	`(1 + 2) * 3`
conversions (cast)	`(int)(a / 2.0)`
alternatives	`(a > 0) ? a : -a`
affectations	`a = 2`
incrémentations	`a++`, `--a`
affectations combinées	`x += y * 2`
appels de méthode	`obj.méthode(...)`

une variable ici peut être :

- une variable locale : `int i; ... i = 2;`
- un argument formel d'une méthode : `void f(int i) { ... i + 1; }`
- un attribut d'un objet : `obj.attribut`

Types en Java

Deux grandes familles de types :

- types primitifs
  - contiennent des valeurs simples
  - `int`, `float`, `boolean`, etc.
  - passage par valeur
  - valeur par défaut : 0, 0.0, `false`
- types objets
  - types définis par le programmeur
  - `class` : définit des instances de classes
  - `interface` : regroupe des objets obéissants à une interface
  - `enum`, `record` : extensions récentes ("sucre syntaxique")
  - tableaux
  - objets alloués explicitement avec `new`, passés par référence, libérés automatiquement
  - valeur universelle par défaut : `null`

les chaînes sont des objets en Java, pas des types primitifs
chaque type primitif a une classe correspondante : `Integer`, `Float`, ...

Types primitifs et leurs opérateurs

- entiers : généralement `int` (32-bit)
  - valeurs : de -2^{-31} à $2^{31} - 1$
  - il existe aussi : `long` (64-bit), `short` (16-bit), `byte` (8bit), toujours signés
  - constantes littérales : `0`, `12`, `0xa0`, `0b001`
  - opérateurs : `+`, `-`, `~`, `*`, `/`, `%`, `<<`, `>>`, `>>>`, `&`, `|`, `^`
- caractères : `char`
  - valeurs : Unicode 16-bit
  - constantes littérales : `'a'`, `'\u03A9'`, `'\''`
- flottants : `float` (32-bit) ou `double` (64-bit)
  - valeurs : flottants à la norme IEEE 754
  - constantes littérales : `0.1`, `1.2e3`, `0.1d`, `0.1f`
  - opérateurs : `+`, `-`, `*`, `/`
- booléens : `boolean`
  - valeurs et littéraux : `true`, `false`
  - opérateurs booléens : `!`, `&&`, `||`
  - comparaisons : `==`, `!=`, `>`, `<`, `>=`, `<=`
  - opérateur ternaire : `?:`

conversion automatique si nécessaire : entier → flottant, petit entier → grand entier, etc.
mais pas de conversion automatique en booléen, contrairement au C
ni d'un type plus large vers un type plus petit : flottant → entier

Déclarations, blocs, affectation

- **déclaration**
  définit des nouvelles variables, avec leur type et valeur initiale (optionnelle)
  - déclaration simple :
    ```
    type var1, var2, ...;
    ```
  - déclaration avec initialisation :
    ```
    type var1 = expr1, var2 = expr2, ...;
    ```
  - déclaration avec inférence de type (Java $\geq$ 10) :
    ```
    var var1 = expr1, var2 = expr2, ...;
    ```

- **affectation**
  change la valeur d'une variable existante (ou celle d'un attribut d'un objet)
  ```
  var = expr;
  ```
  - la valeur stockée doit être compatible avec le type de la variable
  - conversion possible : `var = (type) expr;`
  - certaines conversions sont implicites (int $\rightarrow$ float)

- **bloc**
  ```
  { instruction1; ... ; instructionN; }
  ```
  - regroupe des instructions
  - restreint la portée des variables qui y sont déclarées

Structures de contrôle

- tests
  - `if (expr) instruction`
  - `if (expr) instruction else instruction`
- boucles
  - `while (expr) instruction`
  - `do instruction while(expr)`
  - `for (expr; expr; expr) instruction`
  - `for (type var = expr; expr; expr) instruction`
  - `for (type var : collection) instruction`

  peuvent être préfixées d'un label `L:`
- sortie de boucle ou de switch

  `break;` `continue;`
- sortie de méthode (avec ou sans valeur de retour)

  `return;` `return expr;`
- analyse par cas (en fonction de la valeur d'un entier, d'une chaîne, d'une énumération)
  - `switch (expr) { case expr: inst; break; ... default: insts }`

Instructions (exemples)

factorielle v1

```
int fact(int n) {
  int i = 0, x = 1;
  while (i < n) {
    i = i + 1;
    x = x * i;
  }
  return x;
}
```

factorielle v2

```
int fact(int n) {
  int x = 1;
  for (int i = 1; i <= n; i++)
    x *= i;
  return x;
}
```

switch

```
if (x < 0) x = -x;
switch (x) {
  case 0: y = 1; break;
  case 1: y = 2; break;
  default: y = 0;
}
```

break, continue

```
for (i = 0; i < n; i++)
  for (j = 0; j < m; j++) {
    if (a[j] == 0) break;
    if (a[j] <= a[j]) continue;
    ...
  }
```

Premier exemple de classe : le point

La classe `Point` décrit comment créer et manipuler des points dans le plan.

Invariant à maintenir : les points sont toujours à coordonnées positives.

Point.java

```
package pobj.cours1;
import java.lang.Math;

public class Point {

  private double x,y;

  public Point() {
      x = 100;
      y = 100;
  }
  public Point(double x, double y) {
      this.x = (x < 0) ? 0 : x;
      this.y = (y < 0) ? 0 : y;
  }

  public double getX() { return x; }
  public double getY() { return y; }
```

Point.java (suite)

```
  private void set(double newX, double newY) {
      if (newX >= 0) x = newX;
      if (newY >= 0) y = newY;
  }

  void translate(double mx, double my) {
      set(getX() + mx, getY() + my);
  }

  public double length() {
      return Math.sqrt(x*x + y*y);
  }

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

Points : déclaration, attributs, *getters*

Point.java (début)

```
public class Point {                          // déclaration de classe
    private double x,y;                       // attributs
    public double getX()  { return x; }       // méthode d'accès
    public double getY()  { return y; }       // méthode d'accès
```

Déclaration de classe :

- le code source d'une classe `Toto` doit obligatoirement se trouver dans un fichier du même nom : `Toto.java`
- une seule classe par fichier Java ! (exceptions : classes privées, classes internes)
- par convention, les noms de classe commencent par une majuscule

Attributs :

- un point a deux attributs flottants : `x`, `y`
- chaque instance de la classe `Point` maintient sa valeur des attributs
- les attributs sont privés : `private x,y`
  non directement accessibles en dehors de la classe avec la notation `.x` et `.y`
- mais accessibles en lecture par des méthodes : `public double getX()`

Points : constructeurs

Point.java

```
// constructeur
public Point() {
  x = 100;
  y = 100;
}
```

Point.java

```
// constructeur
public Point(double x, double y) {
  this.x = (x < 0) ? 0 : x;
  this.y = (y < 0) ? 0 : y;
}
```

Un constructeur indique comment initialiser un objet de classe `Point`

- le nom du constructeur est celui de la classe
- il peut avoir des arguments, mais pas de valeur de retour (modification en place)
- on peut définir plusieurs constructeurs (avec des listes d'arguments différentes)

Note : résolution des identifiants

- `obj.x` accède à l'attribut `x` de l'objet `obj`
- `this` dénote l'objet courant
- par défaut, `x` dénote l'attribut `x` de l'objet courant
  $\Longrightarrow$ `x` et `this.x` sont alors synonymes
- `x` peut être masqué par une variable locale, ou un argument de méthode ou constructeur
  $\Longrightarrow$ nous devons utiliser alors `this.x` pour accéder à l'attribut

Points : construction, new

Point.java

```
// constructeurs
public Point() {
  x = 100;
  y = 100;
}
public Point(double x, double y) {
  this.x = (x < 0) ? 0 : x;
  this.y = (y < 0) ? 0 : y;
}
```

client de Point

```
Point p = new Point();

Point q;
q = new Point(10, 10*2);
...
q = new Point(1, 2);
```

L'instruction new crée une nouvelle instance de la classe Point :

- retourne un nouvel objet de type Point
- doit être stockée dans une variable déclarée avec le type Point
  ou un type compatible avec Point (voir suite du cours)
- effectue un appel au constructeur défini dans la classe
- le constructeur appelé est choisi en fonction du nombre et du type des arguments
  cas particulier de surcharge, étudié en détails au cours 7

Points : méthodes

Point.java

```
private void set(int newX, int newY) {
    if (newX >= 0) x = newX;
    if (newY >= 0) y = newY;
}

void translate(double mx, double my) {
    set(getX() + mx, getY() + my);
}
```

client de Point

```
Point p = new Point();
p.translate(-100,100);
```

Appel de méthode :

- `obj.méthode(expr1,...,exprN)`
- `méthode(expr1,...,exprN)`
  est équivalent à `this.method(expr1,...,exprN)`

Visibilité :

- `private set(...)` est une méthode à usage interne, cachée des autres classes
- `public getX()` est une méthode exportée aux autres classes
- `translate(...)` est une méthode exportée aux classes du même package
  par défaut, si ni `public` ni `private` n'est précisé

Diagramme de classe UML

Point
− x : double − y : double
+ Point() + Point(double, double) + getX() : double + getY() : double − set(double, double) : void ∼ translate(double, double) : void + toString() : String

Description d'une classe, en 3 blocs :

1. nom de la classe
2. liste des attributs avec type
3. liste des méthodes et constructeurs
   avec type de retour,
   type et éventuellement nom des arguments

Visibilité :

- + : publique (mot-clé `public`)
- − : privée (mot-clé `private`)
- ∼ : package (absence de mot-clé)

UML : *Unified Modeling Language*

- notation pour la modélisation orientée objet
- diagramme de classes : décrit graphiquement les classes et leurs relations
  en faisant abstraction de l'implantation et du langage
- standardisé : *lingua franca* du développement logiciel

Agrégation : la classe Rectangle

Agrégation :

Un rectangle est composé de deux points : ses coins.

$\Longrightarrow$ les opérations sur les coins d'un `Rectangle` sont déléguées à la classe `Point`.

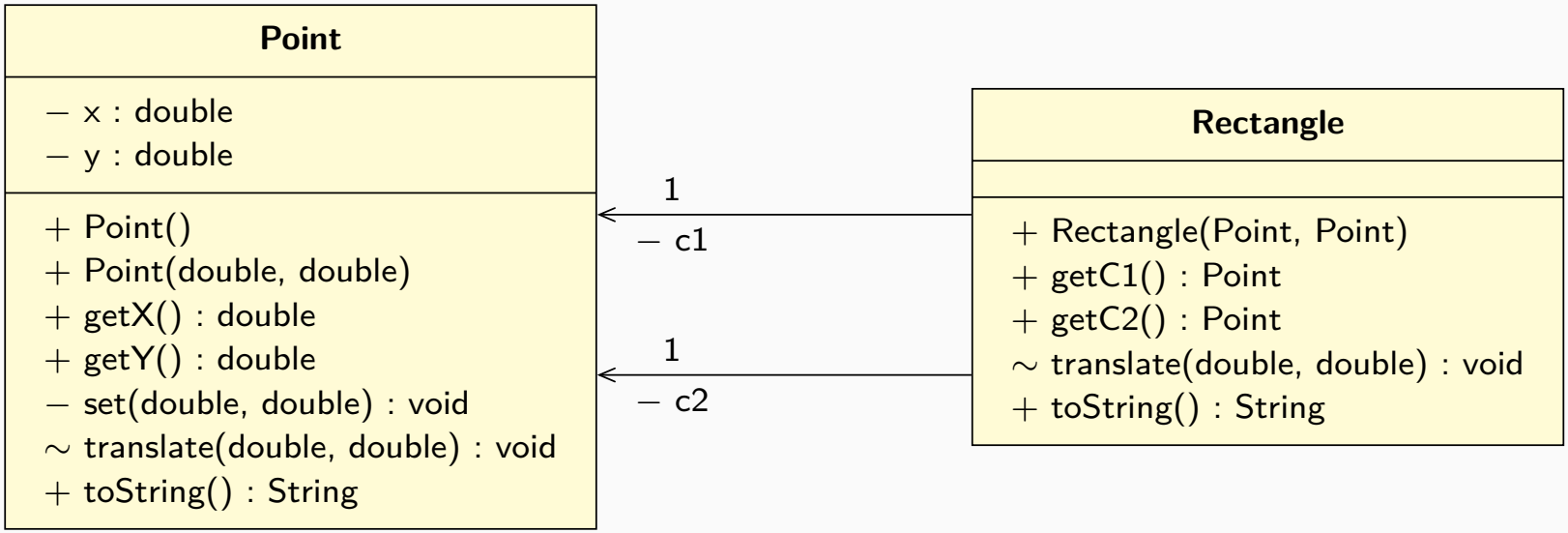
Rectangle.java

```
package pobj.cours1;

public class Rectangle {

    private Point c1,c2;

    public Rectangle(Point c1, Point c2) {
        this.c1 = c1;
        this.c2 = c2;
    }

    public Point getC1() { return c1; }
    public Point getC2() { return c2; }
```

Rectangle.java (suite)

```
    void translate(double mx, double my) {
        c1.translate(mx, my);
        c2.translate(mx, my);
    }

    @Override public String toString() {
        return c1 + "x" + c2;
    }
}
```

Diagramme de classes UML avec association



Pour matérialiser l'agrégation, UML utilise une association :

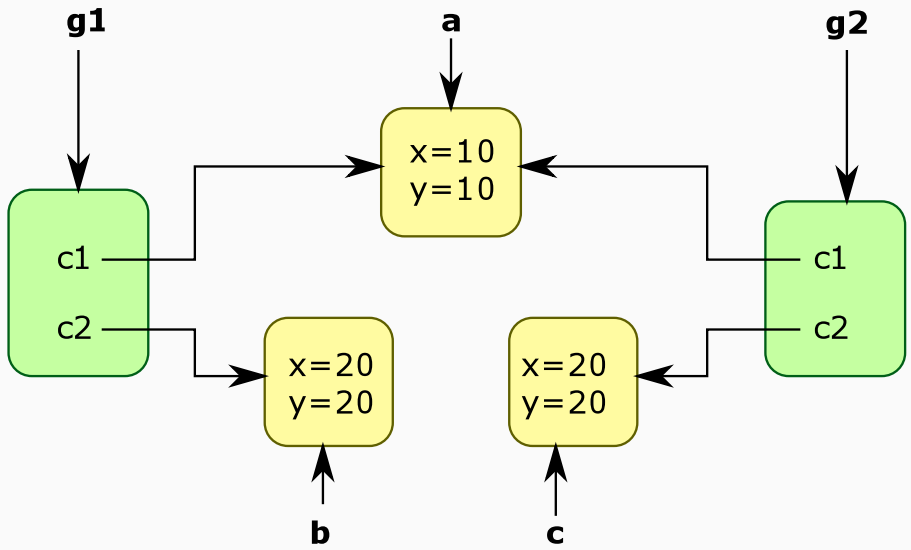
- flèche de la classe client vers la classe utilisée : `Rectangle` → `Point`
- étiquetée par le nom des attributs avec leur visibilité : − `c1`, − `c2`
- étiquetée également par la multiplicité :
  1 ici car chaque attribut `c1`, `c2` dénote un unique `Point`
  la multiplicité * sera utilisée pour les attributs tableaux

Attention : un attribut apparaissant dans une association ne doit pas être aussi présent dans le bloc des attributs de la classe !

Objets, références, égalité physique ==

client

```
Point a = new Point(10,10);
Point b = new Point(20,20);
Point c = new Point(20,20);
Rectangle g1 = new Rectangle(a,b);
Rectangle g2 = new Rectangle(a,c);
```



Les objets sont passés **par référence** :

- `new Point` crée un nouvel objet
- l'affectation `a = ...` stocke une référence sur l'objet dans `a`
- l'appel à `Rectangle(a,b)` passe une référence sur l'objet au constructeur
- le constructeur stocke une référence sur l'objet dans l'attribut `c1`

`a`, `g1.c1` et `g2.c1` pointent sur le même bloc mémoire

$\Longrightarrow$ `g1.translate` va modifier `a.x` et `a,y`, donc changer aussi `g2`

Opérateur d'égalité == : teste si deux références pointent sur le même objet

`a == g1.getC1()` et `a == g2.getC1()`, mais `b != c`

Attributs immuables : mot-clé `final`

Le mot-clé `final` indique qu'un attribut est immuable : (constant)

- il peut être initialisé dans le constructeur
- il ne peut pas être modifié dans les méthodes

Le compilateur vérifie que les attributs `final` ne sont pas modifiés.
Utiliser `final` permet donc d'éviter certaines erreurs de programmation !

version immuable du Point

```
class ConstantPoint {

    final private double x, y;

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

    public double getX() { return x; }
    public double getY() { return y; }

    // pas de méthode set ou translate possible !
}
```

Méthodes et attributs statiques : mot-clé `static`

- attribut `static` : partagé par toutes les instances de la classe
  sans `static`, chaque instance d'une classe a sa version de l'attribut
- méthode `static` : non attachée à une instance de la classe : pas de `this`
  ne peut donc accéder qu'aux attributs et méthodes statiques de la classe !
- syntaxe : `Classe.attribut`, `Classe.méthode(...)`
- exemples : `Math.PI`, `Math.sqrt(2.)`, `System.out.println("Hello")`
- en UML : les attributs et méthodes `static` sont soulignés

compteur décroissant

```
class CountDown {
    static private int nb = 100;
    private int val;

    public CountDown()                { val = nb; if (nb > 0) nb--; }

    static public int getNb()         { return nb; }
    public int getVal()               { return val; }
}
```

Packages, mots-clés `package` et `import`

Java organise les classes en packages, de manière hiérarchique.

Package = chemin `nom1.nom2.···.nomN`

Notre classe `Point` :

- appartient à un package : `package pobj.cours1` (première instruction du fichier)
- a pour nom relatif `Point` et pour nom absolu `pobj.cours1.Point`
- peut être référencée directement par les autres classes du même package
- peut être référencée en dehors du package si
  - `import pobj.cours1` est spécifié en début de fichier
  - ou le nom absolu `pobj.cours1.Point` est utilisé

  et la visibilité de la classe est `public`
- le source de `Point` doit être stocké dans `pobj/cours1/Point.java`

`pobj/cours1/Point.java`

```
package pobj.cours1;
import java.lang.Math;
public class Point {
  ...
    return Math.sqrt(...);
  ...
```

`pobj/cours1/Rectangle.java`

```
package pobj.cours1;
public class Rectangle {
  private Point c1, c2;
  ...
```

`pobj/cours2/Cercle.java`

```
package pobj.cours2;
public class Cercle {
  private pobj.cours1.Point c;
  ...
```

Visibilité et packages

Visibilité des attributs, méthodes et constructeurs,
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`	✓	×	×	×

Mais un attribut privé peut être rendu accessible via des méthodes publiques !

Nous verrons la notion de sous-classe au prochain cours...

Référence null

exemple

```
public class Test {
  private Point a;
  public Test() { }

  public double getX() {
      if (a == null) return 0.0;
      return a.getX();
  }
}
```

`null` représente une référence à un objet inexistant

- `null` peut être stocké dans une variable de tout type objet
- un attribut non fixé par le constructeur est égal à `null` par défaut
- un test `== null` permet de vérifier si une référence est valide
- tout accès (attribut ou méthode) à une référence `null` est **une erreur**

```
Exception in thread "main" java.lang.NullPointerException
```

`null` peut être utile pour représenter une valeur optionnelle
mais `null` est surtout dangereux et source de nombreuses erreurs !

Tableaux

- déclaration (avec initialisation optionnelle)

  ```
  type[] variable;
  type[] variable = { expr1, ..., exprN };
  ```

  où `type` est un type primitif (`int`, ...) ou un type objet

  sans initialisation, le tableau vaut `null` (comme tout objet...)

- création : `new type[expr]`

  où la valeur de l'expression `expr` fixe la taille du tableau

  les éléments sont tous initialisés à `null`, `0` ou `false`

- accès : `variable[expr]`

  `expr` est l'indice, de 0 à la taille - 1

- taille : `variable.length` (syntaxe d'attribut)

  la taille d'un tableau est constante, fixée à la création ou à l'initialisation

- tableaux multidimensionnels : `type[][] v = new type[expr][expr];`
  tableaux de tableaux, vus plus en détails en TME

exemple

```
Point[] points;                 // points est null
points = new Point[12];         // points[0] est null
points[0] = new Point(10,20); // points[0] est un point
points[0].translate(10,10);
```

Chaînes de caractères : classe `String`

En Java, les chaînes de caractères sont des objets de classe `String` (non primitif).

ou, plus précisément, `java.lang.String`

- constantes littérales : `"toto"`, `"Hello\nWorld!"`
- opérateur de concaténation : `+`
- taille d'une chaîne : `chaîne.length()`
- comparaison de chaînes : `chaîne1.equals(chaîne2)`

  ne pas utiliser `==` : deux objets distincts peuvent avoir le même contenu !
- affichage d'une chaîne : `System.out.println(chaîne)`

En Java, les chaînes sont **immuables** :

- il est impossible de modifier le contenu d'une chaîne
- mais une variable peut être modifiée pour pointer sur une nouvelle chaîne. . .

exemple

```
String x = "42";
String y = x;      // x et y référencent la même instance
x = x + "1";       // création d'une nouvelle chaîne "421"
                   // x a changé, mais pas y
```

Méthodes standard, classe `Object`, `@Override`

pobj1/cours1/Point.java

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

Toute classe a des méthodes standards, avec une implantation par défaut :

- `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 possible de redéfinir le comportement de ces méthodes ;
la redéfinition est matérialisée par l'annotation `@Override`.

La classe `Object` dénote un objet générique
sans attribut, avec seulement les méthodes prédéfinies.

Le type `Object` est compatible avec tous les types de classes (voir cours suivant).

Point d'entrée : la méthode `main`

Un programme Java est un ensemble de classes.

L'exécution débute par la méthode `main` d'une classe :

```
public static void main(String[] args)
```

- `public` pour être visible
- `static` car aucun objet n'est encore créé (pas de `this` au démarrage du programme)
- `String[] args` : tableau listant les arguments passés en ligne de commande
  (nous verrons les tableaux et les chaînes de caractères un peu plus loin)

Il peut y avoir plusieurs classes avec chacune sa méthode `main` !

pobj/cours1/Programme1.java

```
package pobj.cours1;

class Programme1 {
    public static void main(String[] args) {
        Point p1 = new Point(10, 10);
        Point p2 = new Point(20, 20);
        Rectangle r = new Rectangle(p1, p2);
        System.out.println("Rectangle: " + r);
    }
}
```

Compilation et exécution en ligne de commande

Compilation : `javac` : compilation de `.java` (source) en `.class` (code-octet)

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

Génère les fichiers : `Point.class`, `Rectangle.class` et `Programme1.class`.
(également possible de passer plusieurs fichiers `.java` à la commande `javac`)

Le répertoire destination, `pobj/cours1`, est déduit des instructions `package pobj.cours1` dans le source (pas du répertoire du code source).

Les classes référencées par `Programme1` doivent être compilées avant `Programme`
(donc dans l'ordre : `Point` → `Rectangle` → `Programme`)

Exécution : `java` : exécution du code-octet

```
java pobj.cours1.Programme1
```

- utilise la notation des package `.`, pas celle des répertoires
- `pobj/cours1/Programme.class` doit exister
- `Programme` doit avoir une méthode `public static void main(String [])`

Introduction aux interfaces

Une interface décrit de manière abstraite
les méthodes devant être implantées *a minima* par une classe.

Une interface contient : des signatures de méthodes publiques
et omet les détails d'implantation :

- le code des méthodes (exception : méthodes *default* vues plus tard)
- les méthodes privées (idem)
- les constructeurs (les interfaces ne sont pas instanciables)
- les attributs (sauf les constantes, déclarées `static final`)

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

pobj1/cours1/IPoint.java

```
public interface IPoint {
    public double getX();
    public double getY();
    public double length();
    public String toString();
}
```

pobj1/cours1/Point.java

```
public class Point implements IPoint {
    private double x,y;
    public Point() { ...
    public double getX() { ...
    public double getY() { ...
    private void set(double ...
...
```

(plus sur les interfaces dans le prochain cours)

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)

En supplément : quelques liens bibliographiques

En ligne :

- Tutoriels sur le site Oracle, jusqu'à Java 8 :
  https://docs.oracle.com/javase/tutorial
  $\Longrightarrow$ à lire si vous n'avez jamais suivi de cours de Java !

- MOOC « Introduction à la programmation orientée objet (en Java) »
  Sam & Chappelier (EPFL), sur Coursera
  https://www.coursera.org/course/intropoojava

Livres :

- Java in a Nutshell
  Ben Evans, David Flanagan
  la 7ème édition couvre jusqu'à Java 11

- The Java™ Programming Language, 4th Edition
  Ken Arnold, James Gosling, David Holmes
  par les auteurs du langage, mais un peu vieux : seulement jusqu'à Java 5

- Design patterns : tête la première
  Eric Freeman, Elisabeth Freeman, Kathy Sierra