Les bases du langage Java

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 1
19 janvier 2016

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 !
   - découverte de traits non spécifiques à la POO
     souvent ajoutés récemment
     aspects fonctionnels, génériques, etc.
3. s'initier aux pratiques de la conception orientée objet
   modélisation UML, *design patterns*

Pour cela nous allons :

- écrire du code Java pour réaliser une application complexe
  projet, réalisé par étapes en TME
- concevoir des morceaux d'application (sur le papier)
  en TD

Organisation de l'UE

Responsable de l'UE : Olivier Sigaud.

Chargés de cours :

- Antoine Miné
- Yann Thierry-Mieg

Chargés de TD et TME :

1. Antoine Miné
2. Olivier Sigaud
3. Loïc Girault
4. Tewfik Ziadi
5. Nicolas Thome
6. Yann Thierry-Mieg

Site :

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

transparents de cours, TD, TME, annales, ... mis a jour régulièrement

Environnement logiciel

- Java 8
  diffusé par Oracle

  JDK téléchargeable sur
  `http://www.oracle.com/technetwork/java/javase/downloads`

- la ligne de commande
  pour compiler et exécuter du Java

- l'environnement de développement (IDE) Eclipse

  `http://www.eclipse.org`

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

Bibliographie et liens

En ligne :

- Tutoriels sur le site Oracle :
  https://docs.oracle.com/javase/tutorial
- Référence de l'API Java 8 :
  https://docs.oracle.com/javase/8/docs/api/
- 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, 6th Edition
  Benjamin J Evans, David Flanagan
  dernière édition, couvrant jusqu'à Java 8
- The Java™ Programming Language, 4th Edition
  Ken Arnold, James Gosling, David Holmes
  par les auteurs du langage, mais seulement jusqu'à Java 5...
- Design Patterns : Elements of Reusable Object-Oriented Software
  Gamma, Helm, Johnson, Vlissides
  livre fondateur !

Évaluation

- examen final : 60 %

  épreuve sur papier, de 2h

- partiel : 25 %

  épreuve sur machine, de 2h

  séance de TME spéciale d'entraînement à l'environnement utilisé au partiel

- contrôle continu : 15 %

  projet, réalisé par étapes en TME

  - en binôme
  - rendu progressif, après chaque TME lié au projet

    rendu immédiat par email au chargé de TME
    + rendu différé d'une semaine (pour compléter vos réponses)
  - en fin de semestre : rendu final et rapport final

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 :

- 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. réutilisabilité
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

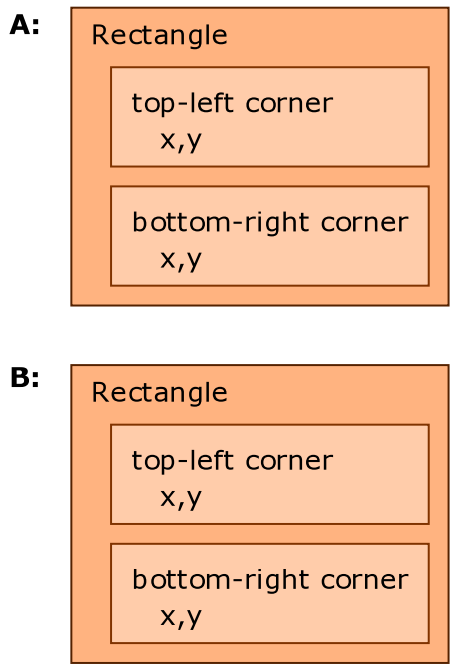
Illustration : fil conducteur

Nous illustrions ces concepts sur un exemple simple,
une application manipulant des objets géométriques : carrés, cercles ;
par exemple, un logiciel de dessin vectoriel (Adobe Illustrator, inkscape)

Contre-exemple :

dans un langage impératif procédural à la C,
le programme serait composé de :
– variables
– structures de données
– procédures et fonctions

Ce n'est pas l'approche objet...



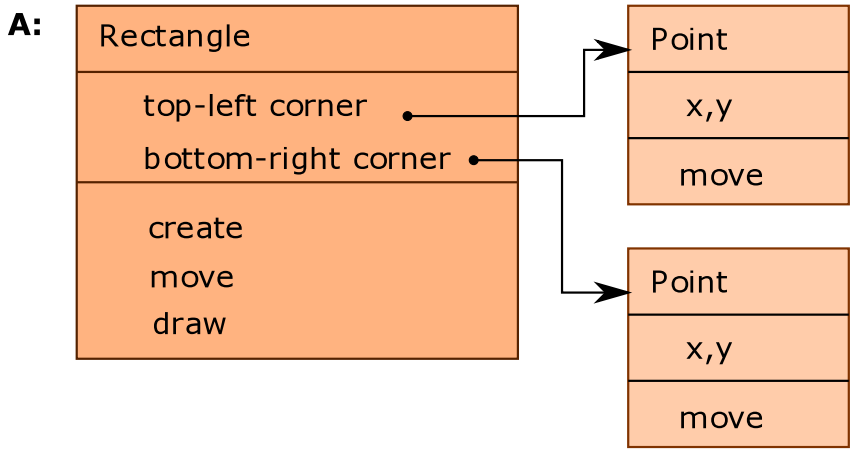

```
Rectangle create_rectangle(...)
void move_rectangle(Rectangle s,...)
void draw_rectangle(Rectangle s, Window, w)
...
```

Concept objet 1 : Encapsulation

Principe :

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
rectangles et points peuvent être bougés ; bouger un rectangle revient à bouger ses coins

Mécanismes Java :

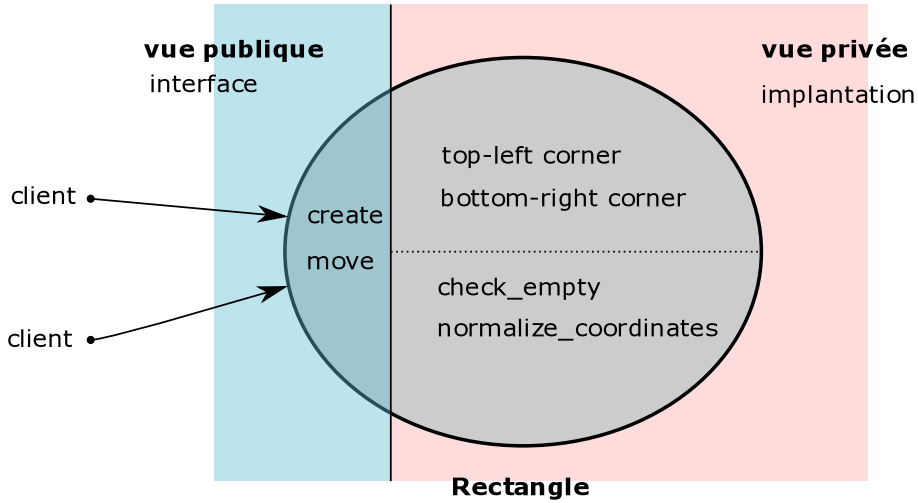
- les **classes**, décrivant des objets ayant les même attributs et méthodes ;
- et, à une granularité plus élevé, les *packages* (organisation hiérarchique).

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

Concept objet 2 : Abstraction

Principe : distinguer

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



Bénéfices :

- meilleur compréhension et utilisation des objets, en ignorant les détails
- se concentrer sur les fonctionnalités des objets, pas leur représentation
- 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

Principe :

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

Mécanismes Java :

- les classes : tous les objets d'une même classe ont les mêmes méthodes
- l'héritage : redéfinir des comportements
- les références : déléguer à un au plusieurs autres objets
  Java n'offre que l'héritage simple, la délégation est donc plus flexible

Concept objet 4 : Polymorphisme

Principe :

- un même objet peut être utilisé dans plusieurs contextes
  un carré peut être utilisé partout où un rectangle est attendu
- le comportement d'un agrégat dépend peu des objets agrégés
  pour manipuler une liste de rectangles, inutile de savoir qu'il s'agit de rectangles
- le comportement d'un objet est paramétré par un autre objet
  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
- 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
- Modula 3 (1980s)
- C++ (1983) extension objet de C
- Python (pré-version en 1991)
- Java (1994)
- C# (2000)

Java :

- créé en 1994 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 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 (les variables doivent être déclarées, avec leur type)
  $\Longrightarrow$ garantie statique de sûreté, à la compilation
- gestion automatique de la mémoire (*garbage collector*)
- support pour les exceptions
  $\Longrightarrow$ garantie dynamique de sûreté, à l'exécution
- support pour le polymorphisme
  polymorphisme d'objet, polymorphisme de surcharge, polymorphisme paramétrique
- support pour 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
  - chargement dynamique de classes
  - vérification des classes au chargement
- bibliothèque standard très riche
  client-serveur, internet, *threads*, interfaces graphiques, etc.

Le langage Java : évolutions

Versions du langage :

- JDK 1.0, 1996
- JDK 1.1, 1997 : classes internes, réflection
- JDK 1.2, 1998 : Swing, collections
- JDK 1.3, 2000
- JDK 1.4, 2002
- JDK 5, 2004 : types génériques, autoboxing, énumerations, *for each*
- JDK 6, 2006 : annotations
- JDK 7, 2011 : améliorations mineures : `switch`, inférence de type, exceptions
- JDK 8, 2014 : aspects fonctionnels : lambdas

Mais aussi, enrichissement de la bibliothèque standard :
212 classes pour JDK 1.0 $\longrightarrow$ 4240 pour JDK 1.8

Les bases du langage Java

Expressions

Les expressions Java sont basées sur celles du C :

	grammaire	exemples
constante	littéral	`2`, `1.2`, `true`, `'a'`
variable	nom-de-variable	`a`
opération unaire	`op` expr	`- x`
opération binaire	expr `op` expr	`2 + 2`
parenthèses	`(` expr `)`	`(1 + 2) * 3`
conversion	`(type)` expr	`(int)(a / 2.0)`
alternative	expr `?` expr `:` expr	`(a > 0) ? a : -a`
affectation	variable `=` expr	`a = 2`
incrémentation	variable `opop`, `opop` variable	`a++`, `--a`
affectation combinée	variable `op=` expr	`x += y * 2`

Une variable peut être :

- une variable locale
- un argument formel d'une méthode
- un attribut d'un objet : `expr.attribut`

Types primitifs et 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 unaires : `+`, `-`, `~`
  opérateurs binaires : `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `>>>`, `&`, `|`, `^`
- 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 : `true`, `false`
  opérateurs booléens : `!`, `&&`, `||`
  comparaisons : `==`, `!=`, `>`, `<`, `>=`, `<=`

conversion automatique si nécessaire : entier → flottant, petit entier → grand entier, etc.
mais pas de conversion en booléen (contrairement au C)

Instructions

- déclaration

  ```
  type var1 = expr1, ..., varN = exprN;
  ```

  toute variable doit être déclarée, avec son type ; l'initialisation est optionnelle

- test

  ```
  if (expr) instruction
  if (expr) instruction else instruction
  ```

- bloc

  ```
  { instruction1; ... ; instructionN; }
  ```

- boucle

  ```
  while (expr) instruction
  do instruction while(expr)
  for (expr; expr; expr) instruction
  for (type var = expr; expr; expr) instruction
  ```

  peuvent être préfixées d'un label `L:`

- sortie de boucle (avec label optionnel)

  ```
  break;    continue;    break L;    continue L;
  ```

- sortie de méthode (avec ou sans valeur de retour)

  ```
  return;    return expr;
  ```

- analyse par cas

  ```
  switch (expr) { case expr: instruction ... default: instruction }
  ```

Instructions (exemples)

factorielle

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

factorielle

```
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

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

Un exemple de classe : le point

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(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);
    }

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

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

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

Contrainte : les points sont toujours à coordonnées positives.

Point : attributs

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 !
- par convention, les noms de classe commencent par une majuscule

Attributs :

- chaque point a deux attributs flottants : `x`, `y`
- les attributs sont privés : `private x,y`
- mais accessibles en lecture par des méthodes : `public double getX()`

Point : constructeur

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
- des arguments optionnels, mais pas de valeur de retour (modification en place)
- plusieurs constructeurs peuvent exister !

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
  mais il peut être masqué par une variable locale ou un argument de fonction
  $\Longrightarrow$ nous utilisons alors `this.x`

Point : construction

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);
```

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

- renvoie un nouvel objet de type `Point`
- doit être stocké dans une variable déclarée avec un type compatible
  (les règles de typage seront détaillées au cours prochain. . .)
- effectue un appel au constructeur défini dans la classe
- le constructeur est choisi en fonction du type et du nombre d'arguments

Point : 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);
```

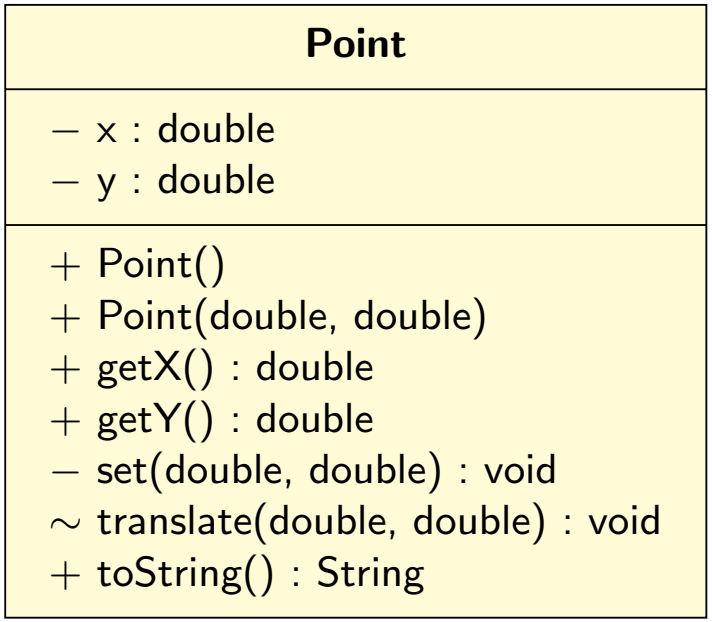
Appel de méthode :

- `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 classes UML



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`)
- $\sim$: 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

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)

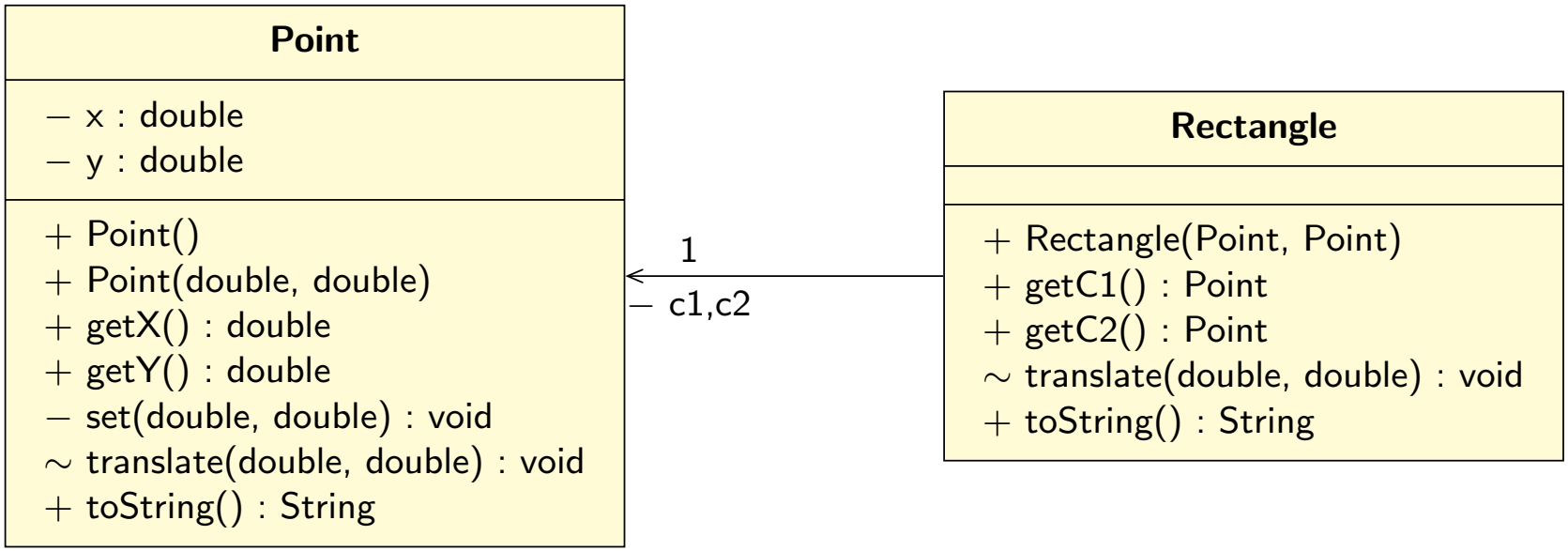
```
    void translate(double mx, double my) {
        c1.translate(mx, my);
        c2.translate(mx, my);
    }

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

Un rectangle est composé de deux points : ses coins.

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

Diagramme de classes UML avec association



Pour matérialiser l'agrégation, UML utilise une association :

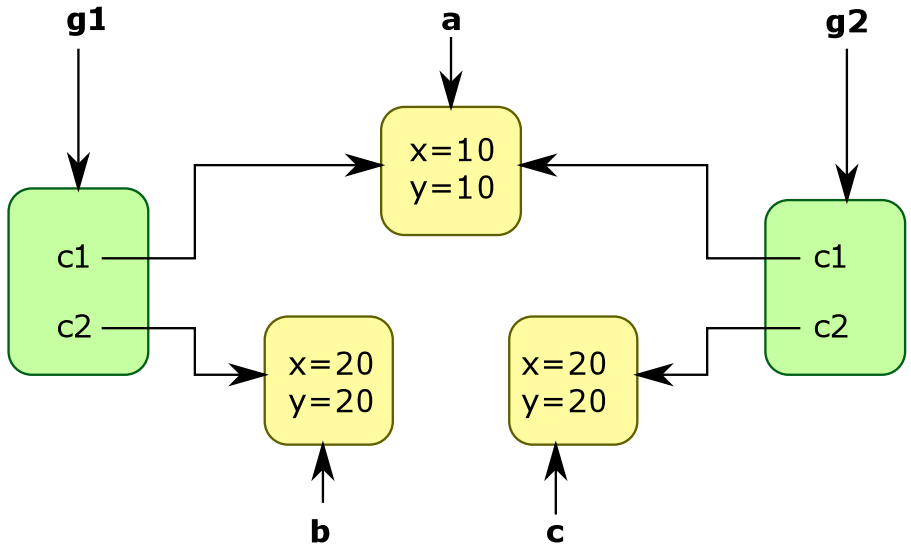
- flèche de la classe client vers la classe utilisée : `Rectangle` → `Point`
- étiquetée par le nom des attributs avec leur visibilité : − `c1`, `c2`
- étiquetée également par la multiplicité :
  1 ici car chaque attribut `c1`, `c2` dénote un unique `Point`
  la multiplicité * sera utilisée pour les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

Le mot-clé `final` indique qu'un attribut est 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 les erreurs de programmation !

version constante du Point

```
class PointConstant {

    final private int x, y;

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

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

Méthodes et attributs statiques : mot-clé `static`

- attribut `static` : partagé par toutes les instances de le classe
  sans `static`, chaque instance d'une classe a sa version de l'attribut
- méthode `static` : non attaché 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 CoundDown()                  { val = nb; if (nb > 0) nb--; }

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

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

Package = chemin `nom1.nom2.···.nomN`
ensemble de classes et de packages, organisés de manière hiérarchique.

Notre classe nommée `Point` :

- appartient à un package : `package pobj.cours1` (première instruction du fichier)
- a pour nom court `Point` et pour nom qualifié `pobj.cours1.Point`
- peut être référencée directement par toutes les autres classes du package
- peut être référencée en dehors du package si
  - `import pobj.cours1` est spécifié en début de fichier, ou
  - le nom qualifié `pobj.cour1.Point` est utilisé
  - et la visibilité est `public`
- le source doit être stockée dans `pobj/cours1/Point.java`

Point.java

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

Rectangle.java

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

Cercle.java

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

Point d'entrée : la méthode `main`

Un programme Java est un ensemble de classes.

L'exécution débute par l'exécution d'une méthode `main` d'une classe :
`public static void main(String[] args)`

- `public` pour être visible
- `static` car aucun objet n'est encore créé
- `String[] args` : 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/Programme1.java
```

génère `pobj/cours1/Programme1.class`
(il est aussi possible de compiler plusieurs classes à la fois)

Exécution : `java` : exécution du code-octet

```
java pobj/cours1/Programme1
```

- `pobj/cours1/Programme1.class` doit exister
- `Programm1` doit avoir une méthode `main`
- toutes les classes référencées par `Programme1` doivent être aussi compilées ; elles seront chargées au fur et à mesure des besoins
- par défaut la classe `a.b.C` est cherchée dans `$PWD/a/b/C.class`
  `-cp` *`path`* permet de spécifier des chemins de recherche additionnels
  `path/a/b/C.class`

Référence `null`

exemple

```
public class Test {
  private Point a, b;
  public Test(Point a) {
      this.a = a;
  }
  public double getRight() {
      if (b == null) return 0.0;
      return b.getX();
  }
}
```

`null` représente une référence à un objet inexistant.

- `null` peut être stocké une variable de tout type classe
- un attribut non fixé par le constructeur sera égal à `null` par défaut
  pour les types primitifs, la valeur 0 ou `false` sera utilisée
- 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 nom de classe

- création : `new type[expr]`

  où `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 l'initialisation

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`.

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
- une variable peut par contre être modifiée pour pointer sur une nouvelle chaîne...

exemple

```
String a = "42";
String b = a;
a = a + "1";      // a changé, b inchangé
```

Méthodes prédéfinies, classe `Object`, `@Override`

pobj1/cours1/Point.java

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

Toute classe a des méthodes prédéfinies, 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, seulement les méthodes prédéfinies.
Le type `Object` est compatible avec tous les types de classe.

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 non-privées

et omet tous les détails d'implantation :

- le code des méthodes
- les constructeurs (les interfaces ne sont pas instanciables)
- les attributs (sauf les constantes, déclarés `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();
    void translate(double mx, double my);
    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 les prochains 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 une complexité algorithmique différente !
- `java.util.List<E>` : type des listes d'éléments de `E`
  utilisation de génériques : polymorphisme paramétrique

Quelques opérations :

- ajout : `boolean add(E)`
- 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)