

LU2IN002 - Introduction à la programmation orientée objet

Responsable de l'UE : Christophe Marsala
(email: Christophe.Marsala@lip6.fr)

Cours du lundi : Sabrina Tollari
(email: Sabrina.Tollari@lip6.fr)

(support réalisé à partir de ceux de Christophe Marsala et de Vincent Guigue)



Cours 1 – lundi 4 septembre 2023

INFORMATIONS

Sur le site Moodle (<https://moodle-sciences-23.sorbonne-universite.fr>)
de l'UE LU2IN002 "Introduction à la Programmation objet - S1-23"

- slides des cours
- version PDF du **Fascicule de TD/TME** (⚠ ne pas imprimer)
 - ◆ Aller chercher la version papier normalement à partir de jeudi soir et **avant votre premier TD/TME**
 - ◆ Où ?
 - Dans les locaux d'ALIAS (Association Ludique et InformAtique de Sorbonne université)
 - Lieu: 14-15/506
 - Horaires habituelles : 12h45 - 13h45 et 18h00 - 19h00
- PDF contenant des **Exercices de TME supplémentaires** pour les rapides en TME
- quizzes d'auto-évaluation en ligne
- forum pour poser des questions
- rendu des devoirs (TME, TME SOLO, projet)
- annales d'exams

PLAN DU COURS

- 1 **Introduction**
 - Philosophie Objet
 - Le langage Java
- 2 **Premier programme**
- 3 **Concepts de base de POO**
- 4 **Guide de survie en Java**

PROGRAMME DU JOUR

1 Introduction

2 Premier programme

3 Concepts de base de POO

4 Guide de survie en Java

FONCTIONNEMENT

- Organisation de l'UE
 - ◆ 1h45 Cours
 - ◆ 1h45 TD : **connaître le cours n avant d'aller en TD n**
 - ◆ 1h45 TME (en binôme sur machine)
- **Evaluation**
 - ◆ contrôle de mi-semestre = 25%
 - ◆ TME solo (en général, pendant la séance de TME9) = 15%
 - ◆ mini-projet (rendu & soutenance lors du dernier TME) = 10%
 - ◆ examen final = 50 % de la note finale

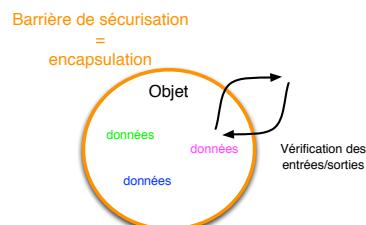
Remarque : en général, les documents (cours, TD, TME, ...) ne sont pas autorisés lors des épreuves

PHILOSOPHIE OBJET

Pourquoi faire de la Programmation Orientée Objet (POO) ?

- Pour développer des systèmes complexes... *Sans se planter*

- ◆ **diviser** le système complexe en une multitude de systèmes simples : **les objets**
- ◆ **sécuriser** l'accès aux données sensibles



- [corollaire] Travailleur à plusieurs... *Avec un minimum de bugs*
 - ◆ toujours penser son programme pour les autres : **sécuriser, simplifier, compartimenter**
 - ◆ double vision : client/fournisseur

JAVA : LE CHOIX D'UNE ARCHITECTURE DYNAMIQUE

De nombreux langages sont orientés objets

★ Le langage Java en est un exemple classique

- langage moderne qui puise son inspiration de sources diverses
 - ◆ syntaxe très proche du C/C++
 - ◆ langage robuste et sûr
 - ◆ fortement typé (vérification forte des types)
- architecture dynamique avec :
 - 1 un compilateur
 - ⇒ langage compilé pour créer des applications performantes
 - 2 et une machine virtuelle, appelée **JVM** (*Java Virtual Machine*), pour exécuter le code Java compilé
 - ⇒ permet la création d'applications indépendantes de la machine physique
- évolution régulière
 - ◆ composants additionnels et fonctionnalités
 - ◆ la syntaxe n'a, elle, que peu évolué
- très utilisé dans l'industrie
 - ◆ API (*Application Programming Interface*) de programmation pour de multiples applications

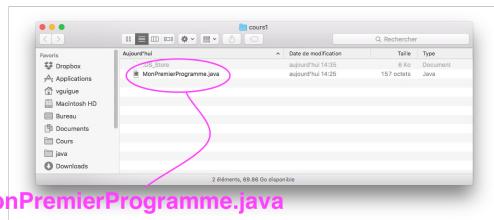
PREMIER PROGRAMME

1 En Java, tout code doit être écrit dans une classe

◆ règle : les noms de classe commencent par une majuscule

```
public class MonPremierProgramme {  
    // code  
}
```

2 1 classe est mise dans 1 fichier du même nom que la classe



3 Programme = ensemble de classes avec un (ou plusieurs) point d'entrée (=main)

◆ ce programme est exécutable après compilation

PREMIER PROGRAMME : COMPILATION/EXÉCUTION

- Pré-requis :

◆ JDK (*Java Development Kit*) installé sur la machine

- ★ JDK = ensemble des bibliothèques et outils pour Java
- ◆ être dans le bon répertoire (!)

1 Compilation

◆ vérification de la syntaxe, droits d'accès...

```
javac MonPremierProgramme.java
```

⇒ Résultat : création d'un fichier exécutable en **bytecode** (code de la JVM) appelé **MonPremierProgramme.class**

- △ L'extension du fichier est **.class**

2 Exécution par la JVM

◆ ici, exécution dans la console

```
java MonPremierProgramme (pas d'extension)
```

⇒ Résultat : affichage dans la console

Bonjour !

PLAN DU COURS

1 Introduction

2 Premier programme

- Programme = ensemble de classes
- Point d'entrée du programme : **main**
- Compilation et exécution d'un programme

3 Concepts de base de POO

4 Guide de survie en Java

PREMIER PROGRAMME : SYNTAXE

La syntaxe des **signatures de classe** et de la **signature du main** est à **apprendre par cœur** (explications dans les cours suivants)

1 Signature d'une classe

◆ **public class NomClasse**

2 Signature d'un main

◆ **public static void main (String [] args)**

△ Aucune modification possible

3 Puis les instructions dans le main

◆ par exemple, affichage de la chaîne "Bonjour !"

Exemple :

```
1 // dans le fichier MonPremierProgramme.java  
2 public class MonPremierProgramme {  
3     public static void main(String [] args) {  
4         System.out.println("Bonjour !");  
5     }  
6 }
```

PLAN DU COURS

1 Introduction

2 Premier programme

3 Concepts de base de POO

- Encapsulation
- Attributs, notamment variables d'instance
- Constructeurs
- Instanciation d'objets
- Représentation mémoire d'un objet
- Méthodes
- Autorisations d'accès : **public/private**
- Représentation UML d'une classe
- Méthodes standards

4 Guide de survie en Java

APPROCHE ORIENTÉE OBJET

Diviser un programme complexe en **objets**

- **objet autonome** : réutilisable dans plusieurs projets
 - ◆ Point, Vecteur, Personne, DisplaySimulation, ...
- **objet sécurisé** : garantie de bon usage par d'autre
 - ◆ un objet intègre des **données**
 - ◆ et des **fonctions/méthodes** pour les manipuler proprement
 - les transactions bancaires sont journalisées,
 - les éléments d'une simulation physique ne se téléportent pas...
- **objet simple et intuitif**

Enjeux

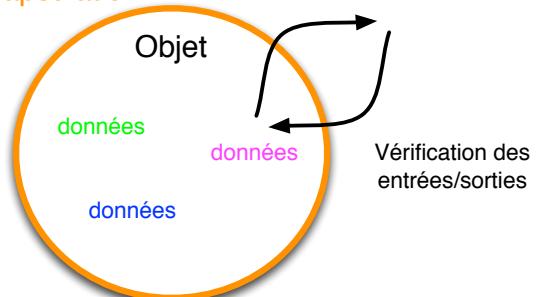
- **réfléchir en amont** au découpage et à la sécurisation
- **documenter le code** (en premier lieu, respecter les conventions pour faciliter la compréhension)

APPROCHE ORIENTÉE OBJET

Barrière de sécurisation

=

encapsulation



APPROCHE ORIENTÉE OBJET : EXEMPLE

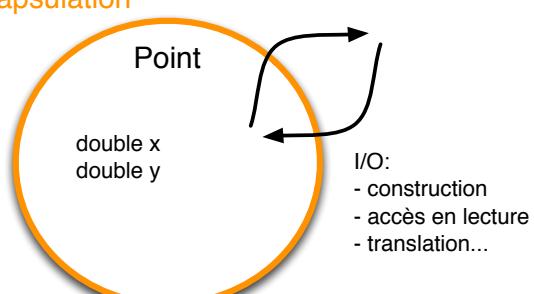
Exemple : un point a des coordonnées (x,y)

On veut pouvoir réaliser certaines opérations de manière sécurisée : créer des points, connaître leur coordonnées, translater des points...

Barrière de sécurisation

=

encapsulation



APPROCHE ORIENTÉE OBJET

Deux visions possibles

- **Le fournisseur** (aussi appelé **développeur**)
 - ◆ celui qui définit le comportement de l'objet
 - ◆ il **définit des classes** d'objets qui comportent des méthodes d'entrée/sortie
- **Le client**
 - ◆ celui qui **utilise des classes** écrites par des fournisseurs pour définir des objets et les manipuler par leurs méthodes d'entrée/sortie
 - ◆ il ne voit que ce que le fournisseur l'a autorisé à voir
- **Remarque** : lorsque l'on conçoit un programme, on est :
 - ◆ fournisseur des classes que l'on définit
 - ◆ mais aussi client des classes que l'on utilise

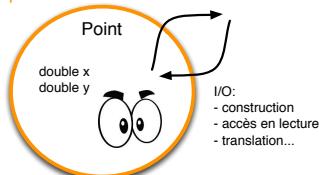
APPROCHE ORIENTÉE OBJET : EXEMPLE

Exemple : un point a des coordonnées (x,y)

Point de vue fournisseur :

- Nouveau fichier, nouvelle classe
 - ◆ Point.java
 - ◆ classe Point
- Comment construire un objet?
 - ◆ attente de 2 valeurs réelles
 - ◆ mise à jour x et y
- Comment établir un dialogue (I/O)?
 - ◆ définition d'une méthode pour observer x

Barrière de sécurisation
=
encapsulation



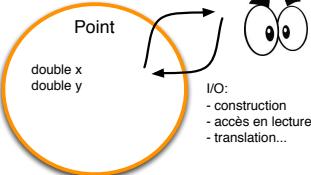
APPROCHE ORIENTÉE OBJET : EXEMPLE

Barrière de sécurisation
=
encapsulation

Exemple : un point a des coordonnées (x,y)

Point de vue client :

- La classe définit un nouveau type de variable
 - ◆ `Point p;`
(la variable p est de type Point)
- Comment construire un objet **Point** ?
 - ◆ donner 2 valeurs réelles + syntaxe
 - ◆ `p = new Point(1.2, 3.1);`
- Comment dialoguer avec un objet **Point** ?
 - ◆ utiliser le `.` pour accéder à l'interface de l'objet
 - ◆ `p.getX();`



SYNTAXE : ATTRIBUTS

- 1 Fichier : 1 classe correspond à 1 fichier
 - ◆ Le nom d'une classe commence par une majuscule
- 2 Déclaration des attributs / champs, dont les variables d'instance
 - ◆ répondre à :
 - ★ De quoi est composé notre objet?
 - ◆ les attributs sont presque toujours private (cf plus loin)
 - ◆ Le nom des attributs commence par une minuscule
- 3 Définir des méthodes (\simeq fonctions)
 - ◆ comment construire un objet?
 - ◆ quelles opérations effectuer sur l'objet?
 - ◆ Le nom des méthodes commence par une minuscule

```
1 // Fichier Point.java
2 public class Point { // classe publique
3   private double x,y; // attributs privés
```

★ x et y sont des attributs de la classe Point, appelés variables d'instance

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 19/47

CRÉER UN OBJET : INSTANCIATION

Point p1 ;

p1 est une variable de type Point (aucun objet n'est créé)

new Point(1,2);

Création d'un objet avec l'opérateur new :

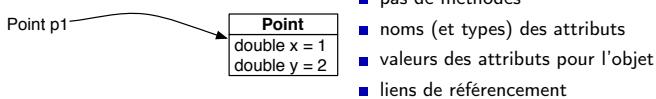
- réservation mémoire pour l'objet
- appel au constructeur pour initialiser x et y

Problème : comment atteindre/manipuler l'objet ?

p1 = new Point(1,2);

La variable p1 référence un objet / une instance de la classe Point

Représentation mémoire correspondante (résultat)

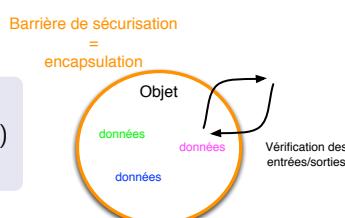


©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 21/47

SYNTAXE : MÉTHODES

Comment manipuler un Point?

- 1 définir des méthodes (\simeq fonctions)
- 2 les invoquer depuis l'extérieur



Fournisseur

permettre l'accès aux attributs (en lecture seulement)

Client

connaître la valeur de x ?

```
1 public class TestPoint{
2   public static void main
3     (String[] args){
4       Point p = new Point(2., 3.1);
5       double px = p.getX();
6       System.out.println(
7         "coord_xudeup:" +px);
8     }
9 }
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 23/47

SYNTAXE : CONSTRUCTEUR

Comment construire un Point?

★ Utiliser un constructeur

⚠️ Attention à considérer le problème des 2 points de vues

Fournisseur

- Besoin : 2 coordonnées fournies en argument
- Action : initialisation des valeurs des attributs

```
1 //Fichier Point.java
2 public class Point{
3   private double x,y;
4   // constructeur
5   public Point(double x2, double y2){
6     x = x2;
7     y = y2;
8   }
9 }
```

Client

ie : utilisateur d'une classe Point existante...

- Besoin : créer un objet Point dans la mémoire avec les bonnes valeurs

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3   public static void main
4     (String[] args){
5       // appel du constructeur
6       // avec des valeurs choisies
7       Point p = new Point(2., 3.1);
8     }
9 }
```

Un constructeur porte le nom de la classe et il ne rend rien !

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 20/47

CRÉER UN OBJET : INSTANCIATION

Côté fournisseur :

mise en route de l'objet

Constructeur = contrat d'initialisation des attributs

```
1 public class Point{
2   private double x,y;
3   public Point(double x2, double y2){
4     x = x2;
5     y = y2;
6   }
7 }
```

Côté client :

création d'un objet

Instanciation = création d'une zone mémoire réservée à l'objet

```
1 public class TestPoint{
2   public static void main
3     (String[] args){
4       // appel du constructeur
5       // avec des valeurs choisies
6       Point p1 = new Point(1,2);
7     }
8 }
```



La variable p1, de type Point, référence une instance de la classe Point dont les attributs ont pour valeur 1 et 2.

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

AUTORISATIONS D'ACCÈS

- **public** : accessible / visible depuis l'extérieur de l'objet (eg : un main, un autre objet...)
- **private** : protégé / invisible depuis l'extérieur de l'objet
 - ◆ ils ont vocation à être appellés depuis l'extérieur
- Les constructeurs et les méthodes sont en général **public**
 - ◆ ils ont vocation à être appellés depuis l'extérieur
- Les attributs sont en général **private**
 - ◆ ils sont protégés et non accessibles depuis l'extérieur

Client

Fournisseur

```
1 public class TestPoint{
2   public static void main
3     (String[] args){
4       private double x,y;
5       public Point(double x2, double y2){
6         x = x2;
7         y = y2;
8       }
9       public double getX(){
10         return x;
11     }
12 }
```

```
1 public class TestPoint{
2   public static void main
3     (String[] args){
4       // opération autorisée
5       Point p = new Point(2., 3.1);
6       // opération autorisée
7       double d1 = p.getX();
8       // opération impossible
9       double d2 = p.x; ERREUR !!!
10    }
11 }
```

Erreur de compilation car l'attribut x est private

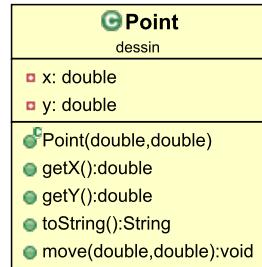
©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 24/47

REPRÉSENTATION UML D'UNE CLASSE

On ne programme pas pour soi-même... mais pour les autres

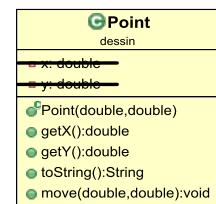
- Respecter les **codes syntaxiques** : majuscules, minuscules...
- Donner des **noms explicites** (classes, méthodes, attributs)
- Développer une **documentation** du code (cf cours javadoc)
- ... Et proposer une **vision synthétique** d'un ensemble de classes : \Rightarrow **UML** : Unified Modeling Language



- nom de la classe
 - attributs
 - méthodes (et constructeurs)
- + code pour visualiser **public**/**private**
+ liens entre classes pour les dépendances (cf cours sur composition)
 \triangle Ne pas confondre : **représentation des classes et représentation des objets en mémoire**

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

25/47



UML CLIENT vs FOURNISSEUR

Plusieurs types de diagrammes pour plusieurs usages :

- Vue **fournisseur** : représentation complète
- Vue **client** : représentation des attributs, constructeurs et méthodes **public** uniquement

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

26/47

REPRÉSENTATION UML (SUITE)

Deux manières de voir l'UML :

- 1 Outil pour une **visualisation** globale d'un code complexe
- 2 Outil de **conception** / développement indépendant du langage

Dans le cadre de LU2IN002 : seulement l'approche ①

Limites de l'UML :

- Vision architecte...
- Mais pas d'analyse de l'exécution du code

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

27/47

MÉTHODES STANDARDS

- Des **méthodes standards** existent et sont utilisables pour tous les objets (cf cours héritage)...
 \diamond mais avec un comportement pas toujours satisfaisant
- Exemple : la **méthode standard String `toString()`** permet de convertir un objet en chaîne de caractères

Client

```
1 Point p = new Point(2., 3.1);
2 System.out.println(p.toString());
```

Affichage :

Point@8764152

Fournisseur

```
3 public class Point{
4 ...
5 // pas de méthode toString() écrite
6 }
```

Fournisseur

```
7 public class Point{
8 ...
9 public String toString(){
10 return "[" + x + "," + y + "]";
11 }
12 }
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

29/47

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

26/47

REFLEXION SUR LA SYNTAXE OBJET

Exemple type : addition de 2 points

Réfléchir à la signature d'une méthode `add` qui crée un nouveau point dont les coordonnées sont les sommes respectives des `x` et `y` de 2 instances de `Point`

Client

```
1 Point p1 = new Point(2., 3.1);
2 Point p2 = new Point(0.5, 1);
3 Point p3 = p1.add(p2);
```

Fournisseur

- En POO, quel(s) est (sont) le(s) paramètre(s) ?
 \triangle Un seul paramètre de type `Point` : addition entre le point courant et le paramètre
- Quel est le type de retour ? `Point`

```
1 public class Point{
2     private double x,y;
3     public Point(double x2,double y2){
4         x = x2;
5         y = y2;
6     }
7     public Point add(Point p){
8         return new Point(x+p.x, y+p.y);
9     }
10 }
```

Syntaxe objet = il faut penser objet... Pas évident au début !

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

COMPILATION / EXÉCUTION

Nous avions vu précédemment comment compiler et exécuter **UNE** classe... Comment faire maintenant qu'il y en a plusieurs?

Syntaxe :

```
> javac Point.java TestPoint.java
> java TestPoint
```

Si on veut compiler tous les fichiers .java du répertoire courant et lancer l'exécution de `TestPoint` seulement si la compilation a réussi :

```
> javac *.java && java TestPoint
```

Remarque : il peut y avoir **plusieurs main**

(mais pas plus de 1 par classe)

- Compilation de tous les `main` d'un coup
- Execution d'un seul (appel à la classe correspondante)

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

30/47

PLAN DU COURS

- 1 Introduction
- 2 Premier programme
- 3 Concepts de base de POO
- 4 Guide de survie en Java
 - Les types de base
 - Conversion entre types et opérateur cast
 - Opérateurs classiques
 - Aléatoire avec `Math.random()`
 - Syntaxe : if, for, while...
 - Durée de vie et visibilité des variables
 - Affichages avec `System.out.println(...)`
 - Chaînes de caractère : `String`
 - Opérateur de concaténation +

TYPES DE BASE DES VARIABLES EN JAVA

- Entier, réel, booléen, caractère : ces types sont disponibles de base en Java avec les opérateurs les plus courants.
- `int, double, boolean, char, byte, short, long, float`
- ⚠ La plupart des types et syntaxes associées sont comparables au C/C++... **Sauf le booléen.**

Le booléen vaut `true/false` et n'est pas convertible en entier

■ Déclarations

```
1 int i; // déclaration de i
2 System.out.println(i); // => 0 (valeur par défaut)
3 double d = 2.6;
4 boolean b = true; // ou false
5 char c = 'a';
```

TYPES DE BASE DES VARIABLES EN JAVA

type		bits	min et max	valeur par défaut
<code>boolean</code>	<code>true/false</code>	1		<code>false</code>
<code>char</code>	<code>Unicode</code>	16	\u0000 à \uFFFF	\u0000
<code>byte</code>	entier signé	8	-128 à 127	0
<code>short</code>	entier signé	16	-32 768 à 32767	0
<code>int</code>	entier signé	32	-2 147 483 648 à +2 147 483 647	0
<code>long</code>	entier signé	64	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807	0
<code>float</code>	réel signé	32		0.0f
<code>double</code>	réel signé	64		0.0d

En général :

- pour les entiers, on utilise `int`
- pour les réels, on utilise `double`

Remarque :

- plusieurs façons d'écrire un `double` : `1.0` ou `1.` ou `1d`
- ⚠ pour écrire un `float`, il faut écrire `1f` ou utiliser un cast : `(float)1.0`



CONVERSIONS ENTRE TYPES

Java, un langage fortement typé

Les types sont très importants en Java : le compilateur vérifie toujours les types des différentes variables

1 Certaines conversions sont **implicites**

Exemples : conversion implicite de `int` vers `double`

```
1 double d1 = 42;
2 int i = 5 ; double d2 = i;
```

2 Certaines conversions doivent être données **explicitement**

Exemple : conversion de `double` vers `int`

- ◆ Risque de pertes d'information : Java ne tolère pas la conversion implicite
- 3 `int i1 = 2.4; ERREUR compilation`
- ◆ Il faut que le programmeur la demande explicitement (pour être sûr que la perte d'information est souhaitée) avec l'opérateur `cast` dont la syntaxe est `(type)`
- 4 `int i2 = (int) 2.4; // OK mais i2 vaut 2 (et non pas 2.4)`

3 Conversions **impossibles**

```
5 int i3 = (int) true; ERREUR compilation
```

OPÉRATEURS CLASSIQUES (PAR ORDRE DE PRIORITÉ)

opérateurs postfixés	[] . expr++ expr--
opérateurs unaires	++expr --expr +expr -expr ~ !
création ou cast	new (type) expr
opérateurs multiplicatifs	* / %
opérateurs additifs	+ -
décalages	<< >> >>>
opérateurs relationnels	< > <= >=
opérateurs d'égalité	== !=
et bit à bit	&
ou exclusif bit à bit	^
ou (inclusif) bit à bit	
et logique	&&
ou logique	
opérateur conditionnel	? :
affectations	= += -= *= /= %= &= ^= = <=>= >>>=

```
1 int i = 5 ;
2 int j = i + 2 * 3; // j vaut 11 car * plus prioritaire que +
3 boolean b = 3 < 4 && false; // b vaut false car < plus prioritaire que &&
4 j += 1; <=> j = j + 1; <=> j++;
5 int k = 1/2; //=0 Attention à la division entière
```

GÉNÉRATION DE NOMBRES ALÉATOIRES

Pour générer un nombre aléatoire, on peut utiliser `Math.random()` qui rend un `double` dans l'intervalle [0, 1[.

Par exemple, pour générer :

■ un réel dans [MIN,MAX] :

```
double val=Math.random()*(MAX-MIN)+MIN;
◆ Exemple : un réel entre 50 (compris) et 200 (non-compris)
double x = Math.random()*150+50;
```

■ un entier dans [MIN,MAX] :

```
int val=(int)(Math.random()*(MAX-MIN+1)+MIN);
◆ Exemple : un entier entre 50 (compris) et 200 (compris)
int y = (int)(Math.random()*151+50);
```

■ un booléen vrai dans 5% des cas

```
boolean onContinue = Math.random()<0.05;
```



CONDITIONNELLES

Syntaxe de l'alternative

```
1 int i=11;
2 if (i > 38) {
3     // code à effectuer dans ce cas
4 } else { // le else est facultatif
5     // Code à effectuer sinon
6 }
```

En cas de clauses multiples

```
1 switch(i){
2 case 1 :
3     // Code à effectuer si i == 1
4     break; // sinon le reste du code est AUSSI effectué
5 case 2 : //
6     // Code à effectuer si i == 2
7     break;
8 default : // Si on n'est passé nulle part ailleurs
9 }
```

STRUCTURES ITÉRATIVES

Même définition des boucles qu'en C/C++

Syntaxes : 2 options (principales) Pour i allant de 0 à 9, faire...

```
1 int i;
2 for(i=0; i<10; i++) {// i prend les valeurs 0 à 9
3                         // ==> 10 itérations
4 // code à effectuer 10 fois
5 }
```

Tant que i inférieur à 10, faire...

```
1 int i = 0;
2 while(i<10) {// i prend les valeurs 0 à 9 =
3                 // 10 itérations
4     // code à effectuer 10 fois
5     i++; // ne pas oublier, sinon boucle infinie !
6 }
```

D'autres syntaxes sont possibles : do...while etc...

INTERRUPTIONS DE FONCTIONS/BOUCLES (1/3)

Trois types d'interruptions de boucles

■ **return** : l'interruption la plus forte. Retour anticipé de la fonction (**sort de la fonction**, pas seulement de la boucle).

```
1 // le modulo par 5 peut-il retourner un entier >=5?
2 public void maFonction(){
3     for(int i=0; i<10; i++){
4         if(i%5>4){
5             System.out.println("C'est très étrange");
6             return;
7         }
8     }
9     System.out.println("L'opération modulo 5 retourne "
10    "toujours un entier inférieur à 5");
11 }
```

INTERRUPTIONS DE FONCTIONS/BOUCLES (2/3)

3 types d'interruptions de boucles

■ **return** ■ **break** : sortie anticipée de la boucle

```
1 // 6 fait-il partie des multiples de 2?
2 public void maFonction(){
3     boolean found = true;
4     for(int i=0; i<10; i++){
5         if(i * 2 == 6){
6             found = true;
7             break; // pas besoin d'aller plus loin
8         }
9     }
10    if(found)
11        System.out.println("6 est un des multiples de 2");
12 }
```

INTERRUPTIONS DE FONCTIONS/BOUCLES (3/3)

3 types d'interruptions de boucles

■ **return**

■ **break**

■ **continue** : passer à l'itération suivante

```
1 // afficher 3./i pour i variant de -10 à 10
2 // il faut penser à sauter le cas 0 qui provoque un problème
3 public void maFonction(){
4     for(int i=-10; i<=10; i++){// -10 et 10 inclus
5         if(i == 0)
6             continue;
7         System.out.println("3./"+i+" = "+(3./i));
8     }
9 }
```

Ces instructions rendent le code plus lisible en limitant notamment le nombre de blocs imbriqués.

DURÉE DE VIE ET VISIBILITÉ DES VARIABLES

Logique de bloc

- les blocs sont repérés par des accolades : {...}
- une fonction, une boucle, une conditionnelle est un bloc

Les variables déclarées dans un bloc sont détruites en sortant du bloc

```
1 public class MaClasse {
2     private int x; // attribut x est visible dans toute la classe
3     public void maMethode() {
4         int i=2; // i locale à la méthode
5         for(int j=0; j<10;j++) { // j existe dans le for
6             if(j%2==0) {
7                 int k = 3; // k existe dans le if
8             }
9             x = 1 ; // OK
10            i = 2 ; // OK
11            j = 3 ; // OK
12            k = 4 ; // ERREUR compilation
13        }
14        x = 5 ; // OK
15        i = 6 ; // OK
16        j = 7 ; // ERREUR compilation
17        k = 8 ; // ERREUR compilation
18    }
19 }
```

AFFICHAGES

Instructions pour afficher dans la console en Java :

■ `System.out.println(...);`

△ le "ln" dans `println` indique un retour à la ligne à la fin

```
1 System.out.println("Bonjour");
2 System.out.println("à tous")
```

Bonjour
à tous

■ `System.out.print(...);` (pas de retour à la ligne)

```
3 System.out.print("Bonjour");
4 System.out.print("à tous")
```

Bonjour à tous

■ `System.out.format(...);`

```
5 String chaine="mon réel";
6 double valDouble=0.8729129425232895;
7 System.out.println(chaine+"="+valDouble);
```

mon réel=0.8729129425232895

```
8 System.out.format("%s=%."3f",chaine,valDouble);
```

mon réel=0,873

©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

43/47

CLASSE String

Gestion des chaînes de caractères

`String` n'est pas un type de base, c'est un objet qui se comporte différemment des types de base... Mais c'est une classe complètement intégrée à Java et son caractère immuable la rapproche très nettement d'un type de base.

⚠ Ne pas confondre l'objet `String` et l'affichage dans la console

Nombreuses méthodes définies dans la classe `String` :

■ extraction de sous-chaînes (`substring`)

■ division en plusieurs chaînes (`split`)

■ recherche de caractères

■ construction de nouvelles chaînes à partir d'expressions régulières (`replace`)...

Toute la documentation sur :

<http://docs.oracle.com/javase/8/docs/api/java/lang/String.html>

©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

45/47

AUTRES EXEMPLES DE FORMATAGES

`String.format(String format, Object... args)`

```
1 double[] tab = new double[10];
2 for(int i=0; i<10; i++)
3     tab[i] = Math.random()*1000;
4 for(int i=0; i<10; i++)
5     System.out.println(tab[i]);
```

1 510.4306229034564
2 775.6503067597263
3 15.528224029893511
4 ...

```
1 for(int i=0; i<10; i++)
2     System.out.println(
3         String.format("%12f", tab[i]));
```

1 510,430623
2 775,650307
3 15,528224
4 ...

```
1 for(int i=0; i<10; i++)
2     System.out.println(
3         String.format("%10.3f", tab[i]));
```

1 510,431
2 775,650
3 15,528
4 ...

Ca marche aussi avec les entiers (%d) et les String (%s)

©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

47/47

CHAÎNES DE CARACTÈRES ET CONCATÉNATION

En Java :

■ le type pour les chaînes de caractères est `String`

■ l'opérateur pour concaténer est `+`

```
1 String s1 = "Luke"; // création d'une chaîne
2 s1 = s1 + "est le frère de Leia"; // + concaténation
3 System.out.println(s1);
```

Luke est le frère de Leia

■ Tout type de base peut être concaténer avec `String`

```
4 double d=3.4; int i=10; char c='A'; boolean b=true;
5 String s2 = "mon message:" + 1.5 + " " + d; // String + double
6 s2 += " " + i; // String + int
7 s2 += " " + c; // String + char
8 s2 += " " + b; // String + boolean
9 System.out.println(s2);
```

mon message :1.5 3.4 10 A true

©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

44/47

CLASSE String (SUITE)

2 choses à retenir sur les `String`

1 Les chaînes sont immutables : modifier une chaîne existante est impossible, il faut créer une nouvelle chaîne qui est une modification de l'ancienne. Cela rend la classe peu efficiente dans certain cas... Et il faut alors se tourner vers des objets plus évolués (`StringBuffer` notamment)

2 Ne pas utiliser `==` avec les `String` mais toujours la méthode `equals`. Les deux versions coïncident mais la première donnera régulièrement des résultats faux (que nous expliquerons plus tard).

```
1 String s1 = "Leia";
2 String s2 = "Luke";
3 if( s1.equals(s2) )
4     System.out.println("les chaînes sont identiques");
5 else
6     System.out.println("les chaînes sont différentes");
```

©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

46/47

LU2IN002 - Introduction à la programmation orientée objet

Responsable de l'UE et du cours du vendredi :
Christophe Marsala
(email: Christophe.Marsala@lip6.fr)

Cours du lundi : Sabrina Tollari
(email: Sabrina.Tollari@lip6.fr)

(support réalisé à partir de ceux de Christophe Marsala et de Vincent Guigue)



Cours 2 – lundi 11 septembre 2023

PLAN DU COURS

1 Rappels et vocabulaire

- Constructeur par défaut
- Accesseurs, mutateurs
- Objet courant
- Conventions d'écritures

2 Surcharge, this...

3 Cycle de vie des objets

PROGRAMME DU JOUR

1 Rappels et vocabulaire

2 Surcharge, this...

3 Cycle de vie des objets

RAPPELS ET VOCABULAIRE : CLASSE, ATTRIBUT

```
1 public class Point {  
2     private double x,y;  
3     public Point(double x2, double y2){  
4         x = x2;  
5         y = y2;  
6     }  
7     public double getX() {  
8         return x;  
9     }  
10    public void setX(double x2) {  
11        x=x2;  
12    }  
13    public String toString() {  
14        return "[" + x + "," + y + "]";  
15    }  
16 }
```

classe \simeq modèle pour créer des objets

Une classe est composée de :

- attributs
- constructeurs
- méthodes

Attributs

Il existe différents types d'attributs.

x et y sont des attributs qui sont aussi appellés variables d'instance

RAPPELS ET VOCABULAIRE : CONSTRUCTEUR

constructeur

- porte le même nom que la classe
- pas de valeur de retour (même pas void)

```
1 public class Point {  
2     private double x,y;  
3     public Point(double x2, double y2){  
4         x = x2;  
5         y = y2;  
6     }
```

Quel est son rôle ?

- initialiser les variables d'instance
- faire d'autres initialisations nécessaires à la création de l'objet

Quand est-il appelé ?

- quand on crée un nouvel objet avec new
- Point p = new Point(1,2);

Si une classe ne contient pas de variables d'instance, il peut être inutile d'écrire un constructeur, mais on peut quand même créer des objets

constructeur par défaut : si une classe ne contient pas de constructeurs, le compilateur ajoute automatiquement un constructeur sans paramètre

Exemple :

```
1 public class A {  
2     // RIEN  
3 }
```

Dans le main d'une classe TestA
A a1 = new A(); // OK

RAPPELS ET VOCABULAIRE : MÉTHODES

méthode \simeq fonction définie dans une classe

Certaines méthodes sont un peu particulières :

- les accesseurs ("getter") dont le seul but est d'accéder à la valeur d'un attribut
 - ◆ Exemple : double getX() { return x; }
 - ◆ Par convention, la signature d'un accesseur est : typeAttribut getNomAttribut()
- les mutateurs ("setter") dont le but est de modifier la valeur d'un attribut
 - ◆ Exemple : void setX(double x2) { x=x2; }
 - ◆ Par convention, la signature d'un mutateur est : void setNomAttribut(typeAttribut nomVar)
- les méthodes standards qui sont déjà définies dans chaque objet, même si elles ne sont pas écrites dans la classe
 - ◆ Exemples : toString, equals...
- la méthode main : point d'entrée du programme

RAPPELS ET VOCABULAIRE : OBJET/INSTANCE

objet = instance d'une classe

```
1 public class Point {  
2     private double x,y;  
3     public Point(double x2, double y2){  
4         x = x2; y = y2;  
5     }  
6     public Point add(Point p){  
7         return new Point(x+p.x, y+p.y);  
8     }  
9 }
```

La variable p1 référence un objet/instance de la classe Point.

La variable p2 référence un autre objet/instance de la classe Point.

⇒ Il y a 2 objets/instances de la classe Point créés

objet courant = l'objet avec lequel on a appelé la méthode

```
23 Point p3 = p1.add(p2);
```

A la ligne 7, l'objet courant est l'objet référencé par p1
(le paramètre p correspond à p2)

```
24 Point p4 = p2.add(p1);
```

A la ligne 7, l'objet courant est l'objet référencé par p2
(le paramètre p correspond à p1)

RAPPELS : CONVENTIONS D'ÉCRITURES

■ Le nom des classes et des constructeurs commence par une majuscule

◆ Exemples : MaClasse, MaClasse()

■ Le nom des méthodes et des variables (dont les attributs) commence par une minuscule

◆ Exemples : maMethode(), maVariable

■ Les mots réservés sont obligatoirement écrits tout en minuscules

◆ Exemples : public, class, true, false...

■ Les constantes sont généralement écrits tout en majuscules

◆ Exemples : Math.PI, MA_CONSTANTE

⇒ Rien qu'à la façon dont c'est écrit, vous pouvez savoir ce que c'est (une variable, une méthode, un constructeur...)

⚠ Bien respecter ces conventions d'écritures quand vous écrivez un programme Java

SURCHARGE DU CONSTRUCTEUR

Comment construire un Point? ... de plusieurs manières !

■ 2 valeurs à fournir : le plus classique

Ex :

■ 1 valeur : affectation de la même valeur pour x et y

■ 0 valeur : initialisation à (0,0) (ou aléatoirement)

⇒ il suffit de définir plusieurs constructeurs (= surcharge)

⚠ signatures toutes différentes

Création de points en appelant le constructeur en fonction des besoins

```
1 public class Point{  
2     private double x,y;  
3     public Point(double x2, double y2){  
4         x = x2; y = y2;  
5     }  
6     public Point(double d){ // surcharge  
7         x = d; y = d;  
8     }  
9     public Point(){ // autre surcharge  
10        x = 0; y = 0;  
11    }  
12}
```

```
31 Point p1 = new Point(2., 3.1);  
⇒ appel constructeur à 2 paramètres  
32 Point p2 = new Point(4.);  
⇒ appel constructeur à 1 paramètre  
⇒ x et y auront la même valeur  
33 Point p3 = new Point();  
⇒ appel constructeur sans paramètre  
⇒ x=0 et y=0
```

VARIABLE D'INSTANCE, LOCALE, PARAMÈTRE

```
1 public class Point {  
2     private double x,y;  
3     public Point(double x2, double y2) {  
4         x=x2; y=y2;  
5     }  
6     public Point add(Point p2) {  
7         Point p3=new Point(x+p2.x, y+p2.y);  
8         return p3;  
9     }  
10 }  
11 }
```

Quelles sont les variables qui sont :

■ des variables d'instance ?

⇒ x, y

■ des paramètres ?

⇒ x2, y2, p2

■ des variables locales ?

⇒ p3

PLAN DU COURS

1 Rappels et vocabulaire

2 Surcharge, this...

- Surcharge de constructeurs
- Surcharge de méthodes
- Le mot clé this

3 Cycle de vie des objets

SURCHARGE DE CONSTRUCTEUR

En général, que doit faire un constructeur ?

★ Initialiser les variables d'instance

★ Si besoin, faire d'autres initialisations

Un point a des coordonnées (x,y) et aussi un nom.

```
1 public class Point {  
2     private double x,y;  
3     private String nom;  
4     public Point(double x2, double y2){  
5         x=x2; y=y2;  
6     }  
7     public Point(double x2, double y2, String n){  
8         x=x2; y=y2;  
9         nom=n;  
10    }  
11    public String toString(){  
12        return "Point "+nom;  
13        +"[" +x+"," +y+"]";  
14    }  
15 }  
16 }
```

Dans le main :

```
21 Point p=new Point(2., 3.1);  
22 System.out.println(p.toString());  
23 }  
24 }
```

Quel est l'affichage ?

"Point null [2.0,3.1]"

Quel est le problème ?

⇒ Dans le constructeur à 2 paramètres, la variable nom n'a pas été initialisée

Bonne pratique

En général, chaque constructeur doit initialiser chaque variable d'instance

```
4 public Point(double x2, double y2){  
5     x=x2; y=y2;  
6     // Nom aléatoire entre "P1" et "P999"  
7     nom="P"+(int)(Math.random()*999+1);  
8 }
```

Affiche : "Point P198 [2.0,3.1]"

SURCHARGE DE MÉTHODE

Surcharge de méthode

- Plusieurs méthodes avec le **même nom**, mais **paramètres différents**
- Le type de retour ne compte pas

```
1 public class Point {  
2     private double x,y;  
3     ...  
4     public void move(double dx, double dy){  
5         x+=dx; y+=dy;  
6     }  
7     public void move(int dx, int dy){  
8         x+=dx; y+=dy;  
9     }  
10    public void move(double dx, double dy,  
11                      double scale){  
12        x+=dx*scale; y+=dy*scale;  
13    }  
14    public void move(Point p){  
15        x+=p.x; y+=p.y;  
16    }  
17 }
```

Rappel : dans la classe, accès total aux attributs privés des autres objets de la **même classe** (=> ligne 14 : p.x et p.y OK)

Pourquoi faire de la surcharge de méthode ?

- ★ Même nom de méthode pour faire la même chose
- Point p=new Point(1,2);
p.move(a,b);
- Que les variables a et b (supposées définies avant) soient de type int ou de type double, l'instruction p.move(a,b) est OK

SURCHARGE DE MÉTHODE

△ Une classe ne peut pas avoir deux méthodes avec la **même signature**

△ Ce qui compte ce sont **le type et l'ordre** des paramètres et non pas **le nom** des paramètres

Quelle est la signature des méthodes move ci-dessous ? Est-ce qu'elles compilent ?

```
1 public class Point {  
2     private double x,y;  
3     ...  
4     public void move(double x2,double y2){//OK move(double,double)  
5         x=x2; y=y2;  
6     }  
7     public void move(double dx,double dy){//ERREUR move(double,double)  
8         x+=dx; y+=dy;  
9     }  
10    public void move(double x2,int y2){//OK move(double,int)  
11        x=x2; y=y2;  
12    }  
13    public void move(int x2,double y2){//OK move(int,double)  
14        x=x2; y=y2;  
15    }  
16 }
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 14/30

LE MOT CLÉ this

this : référence de l'objet courant dans une classe

- this.maVariable : accès à la variable d'instance de l'objet courant appelée maVariable
 - Exemples de syntaxe : this.x, this.y
- this.nomMethode(...) : appel de la méthode nomMethode (de **même signature**) de l'objet courant
 - Exemples de syntaxe : this.toString(), this.add(p);
- this(...) : appel au constructeur (de **même signature**) de l'objet courant
 - Exemples de syntaxe : this(1,5); this(3); this();

Quelques exemples d'utilisation dans les slides suivants.

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 15/30

LE MOT CLÉ this : APPEL AU CONSTRUCTEUR (1/2)

- Problème : quand il y a plusieurs constructeurs, comment éviter de répéter inutilement des instructions ?

```
1 public class Point {  
2     private double x,y;  
3     public Point(double x, double y){  
4         this.x = x;  
5         this.y = y;  
6         long bloc d'instructions d'initialisation  
7     }  
8     public Point(double d){ // surcharge  
9         x = d;  
10        y = d;  
11        même long bloc d'instructions  
12    }  
13    public Point(){ // autre surcharge  
14        x = 0 ;  
15        y = 0 ;  
16        même long bloc d'instructions  
17 }
```

Comment éviter de répéter ce **même long bloc d'instructions d'initialisations** dans chaque constructeur ?

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 17/30

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 14/30

LE MOT CLÉ this : this.maVariable

Pour rendre le code plus lisible, on voudrait que le paramètre qui sert à initialiser une variable d'instance porte le même nom que la variable d'instance

```
1 public class Point {  
2     private double x,y;  
3     public Point(double x, double y) {  
4         x = x; FAUX  
5         y = y; FAUX  
6     }  
7 }
```

△ FAUX, car le **paramètre x cache la variable d'instance x**
⇒ La variable d'instance x n'a pas été initialisée

★ Solution : utiliser **this.x** pour préciser que c'est la variable d'instance x

```
1 public class Point {  
2     private double x,y;  
3     public Point(double x, double y) {  
4         this.x = x;  
5         this.y = y;  
6     }  
7 }
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 16/30

LE MOT CLÉ this : APPEL AU CONSTRUCTEUR (2/2)

- Solution : utiliser un appel à un autre constructeur **this(...)**

```
1 public class Point {  
2     private double x,y;  
3     ...  
4     public Point(double x, double y){  
5         this.x = x;  
6         this.y = y;  
7         long bloc d'instructions d'initialisation  
8     }  
9     public Point(double d){  
10        this(d,d); // appel du constructeur Point(double, double)  
11    }  
12    public Point(){  
13        this(0,0); // appel du constructeur Point(double, double)  
14    } // Autre solution ? this(); // appel du constructeur Point(double)  
15 }
```

★ Le résultat est le même, mais les instructions d'initialisations ne sont plus répétées (code plus lisible, évite les bugs...)

△ **this(...)** doit être la première instruction du constructeur

△ **this(...)** doit toujours être utilisé dans un constructeur, et jamais dans une méthode

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 18/30

LE MOT CLÉ this SEUL

- Dans quel cas utiliser le mot clef **this** seul ?
 - Quand on a besoin de la référence vers l'objet courant

Exemple : additionner un point avec lui-même

```

1 public class Point {
2     private double x,y;
3     public Point(double x2,double y2) {
4         x=x2; y=y2;
5     }
6     public Point add(Point p2) {
7         return new Point(x+p2.x, y+p2.y);
8     }
9     public Point addSelf() {
10        return add(this); // appelle la méthode de signature add(Point)
11    }
12    public String toString() {
13        return "["+x+","+y+"]";
14    }
15 }
```

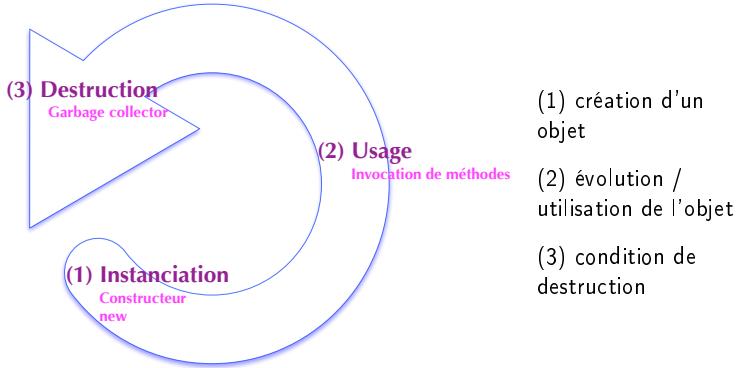
```

16 Point p1=new Point(1,2);
17 Point p2=p1.add(p1);
18 System.out.println(p2.toString()); // Affiche : [2.0.4.0]
19 Point p3=p1.addSelf();
20 System.out.println(p3.toString()); // Affiche : [2.0.4.0]
```

©2021-2022 C. Marsala / V. Guiguer LU2IN002 - POO en Java 19/30

CYCLE DE VIE : DÉFINITION

Se placer du point de vue de l'objet :



©2021-2022 C. Marsala / V. Guiguer LU2IN002 - POO en Java 21/30

(2) USAGE

- le **fournisseur** développe et garantit le bon fonctionnement des méthodes pour *utiliser* l'objet correctement

```

1 public class Point {
2     private double x,y;
3     public Point(double x2,double y2){
4         x = x2; y = y2;
5     }
6     public void move(double dx, double dy){
7         x += dx; y += dy;
8     }
9     ...
10 }
```

- le **client** invoque les méthodes sur des objets pour les manipuler

```

11 Point p1 = new Point(1,2);
12 p1.move(2,3); // p1 => [x=3, y=5]
```

©2021-2022 C. Marsala / V. Guiguer LU2IN002 - POO en Java 23/30

PLAN DU COURS

1 Rappels et vocabulaire

2 Surcharge, this...

3 Cycle de vie des objets

- (1) Création (2) Usage (3) Destruction
- Référence null
- Déréférencement d'un objet
- Logique de bloc
- Garbage collector

(1) INSTANCIATION

Coté client :

création d'un objet

Instanciation = création d'une zone mémoire réservée à l'objet

`Point p1 = new Point(1,2);`

Point p1 → **Point**
double x = 1
double y = 2

Diagramme mémoire
(représentation des objets dans la mémoire)

Coté fournisseur :
mise en route de l'objet

Constructeur = contrat d'initialisation des attributs

```

1 public class Point {
2     private double x,y;
3     public Point(double x2,double y2){
4         x = x2;
5         y = y2;
6     }
7 }
```

RÉFÉRENCE null

- Que se passe-t-il quand on déclare une variable sans l'initialiser avec un objet ?

1 Cas 1 : on n'initialise pas du tout la variable

```
1 Point p; // OK p vaut null
```

```
2 p.move(2,3); ERREUR compilation : variable p might not have been initialized
```

⇒ Le compilateur a détecté l'erreur de programmation

2 Cas 2 : on initialise la variable à null

```
3 Point p = null; // OK utile dans certains cas
```

```
4 p.move(2,3); // OK compilation, mais...
```

```
5 // ERREUR à l'exécution NullPointerException
```

△ `null.method()`; ⇒ erreur `NullPointerException`

Remarque : on peut mettre la valeur `null` à n'importe quel endroit où un objet est attendu... mais cela peut provoquer des crashes !!!

```

10 public class UnObjet {
11     // (classe sans importance)
12     public void maFonction(Point p){
13         ...
14         p.move(2,3); // si p non null
15     }
16 }
```

```

17 // dans le main
18 UnObjet obj = new UnObjet();
19 Point p = new Point(1,2);
20 obj.maFonction(p); // OK
21
22 obj.maFonction(null);
23 // La méthode doit gérer !
```

©2021-2022 C. Marsala / V. Guiguer LU2IN002 - POO en Java

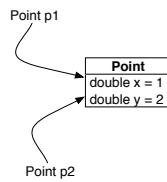
24/30

(3) DESTRUCTION

- 1 Un objet est détruit lorsqu'il n'est plus référencé
- 2 La destruction est implicite (contrairement au C++) et traitée en tâche de fond (garbage collector)
- Un objet peut être référencé plusieurs fois...

```
1 Point p1 = new Point(1,2);
2 Point p2 = p1;
```

Combien d'objets ont été créés ?
Quelle est la représentation mémoire ?



- △ Il y a un seul objet créé, mais deux variables p1 et p2 qui réfèrent à cet objet
- mais quand est-il déréférencé?

RETOUR SUR LA LOGIQUE DE BLOC (1/2)

- △ Le déréférencement dépend de l'endroit où la variable est déclarée, et non pas de l'endroit où la variable est initialisée

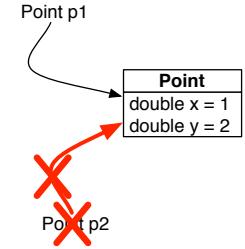
```
1 {
2   Point p1 = new Point(1,2);
3   System.out.println(p1);
4 } // destruction de
5 // la variable p1
6
7 System.out.println(p1);
8 // ERREUR DE COMPILE
9 // p1 n'existe plus ici !
```

```
10 Point p1; // déclaration
11          // avant le bloc
12 {
13   // initialisation de p1
14   p1 = new Point(1,2);
15   System.out.println(p1);
16 } // pas de destruction de p1
17
18 System.out.println(p1);
19 // OK, pas de problème
```

RETOUR SUR LA LOGIQUE DE BLOC (2/2)

- △ Ne pas confondre :
 - ♦ la destruction d'une variable
 - ♦ et la destruction d'un objet

```
1 Point p1; // déclaration
2          // avant le bloc
3 {
4   Point p2 = new Point(1,2);
5   // initialisation de p1
6   p1 = p2;
7   System.out.println(p1);
8 } // destruction de p2
9
10 System.out.println(p1);
11 // OK, pas de problème
12 // p1 existe et l'objet existe
```

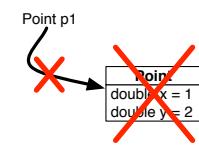


- Fin de bloc = destruction des variables déclarées dans le bloc
- Destruction d'objet ⇔ l'objet n'est plus référencé

DESTRUCTION DES OBJETS

Destruction d'objet ⇔ l'objet n'est plus référencé

```
1 Point p1 = new Point(1,2);
2 p1 = null; // référence vers rien
```



- L'objet n'est plus référencé, il sera détruit automatiquement par le ramasse-miettes (Garbage Collector)
- La variable p1 existe toujours

Remarques :

- En Java, pas besoin d'expliquer comment détruire un objet ≠ C++ définition de destructeurs
- Le Garbage Collector planifie la destruction
- Appel explicite au garbage collector (pour libérer la mémoire) :
`System.gc();`

DESTRUCTION DES OBJETS

... sur un exemple parlant :

```
1 Point p1 = new Point(1,2);
2 Point p2 = new Point(3,4);
3 // Point p3 = p1; // différence avec et sans cette ligne
4 p1 = p2;
```

- Cas 1 : ligne 3 commentée.
 - ♦ l'objet Point(1,2) est détruit à l'issue du re-référencement de l'objet référencé par p1...
 - ♦ ... de toutes façons, cet objet était devenu inaccessible.

```
1 Point p1 = new Point(1,2);
2 Point p2 = new Point(3,4);
3 Point p3 = p1; // différence avec et sans cette ligne
4 p1 = p2;
```

- Cas 2 : ligne 3 dé-commentée
 - ♦ l'objet Point(1,2) est conservé...
 - ♦ on y accède grâce à la variable p3

LU2IN002 - Introduction à la programmation orientée objet

Responsable de l'UE et du cours du vendredi :
Christophe Marsala
(e-mail : Christophe.Marsala@lip6.fr)

Cours du lundi : Sabrina Tollari
(e-mail : Sabrina.Tollari@lip6.fr)

(support réalisé à partir de ceux de Christophe Marsala et de Vincent Guigue)



Cours 3 – lundi 18 septembre 2023

PROGRAMME

1 Remarques diverses

- Classes enveloppes (wrappers)
- Méthode `toString()`
- Représentation mémoire

2 Notion de composition

3 Programmation objet : divers

4 Javadoc, débogage : devenir autonome...

PROGRAMME DU JOUR

1 Remarques diverses

- Classes enveloppes (wrappers)
- Méthode `toString()`
- Représentation mémoire

2 Notion de composition

- UML et composition

3 Programmation objet : divers

- Type de base vs Objet
- Copie d'objet
- Constructeur de copie
- Égalité référentielle
- Égalité structurelle

4 Javadoc, débogage : devenir autonome...

- debugger son programme
- se documenter et documenter soi-même

CLASSES ENVELOPPES

Les types de base en JAVA sont doublés de **wrappers** (ou classes enveloppes) pour :

- fournir quelques outils et constantes fort utiles
- utiliser les classes génériques (cf cours `ArrayList`)

Exemples de classes enveloppes :

`int` → `Integer` `boolean` → `Boolean`
`double` → `Double` `char` → `Character`

Exemples de constantes et méthodes outils de la classe `Double` :

1 `Double d1 = Double.MAX_VALUE; // valeur maximum possible`
2 `Double d2 = Double.POSITIVE_INFINITY; // valeur spécifique`
3 `Double d3 = Double.valueOf("3.5"); // String => double`
Documentation : <http://docs.oracle.com/javase/8/docs/api/java/lang/Double.html>

Exemples de conversions implicites = (un)boxing (depuis JAVA 5)

- autoboxing = conversion automatique type de base vers enveloppe
- unboxing = conversion automatique enveloppe vers type de base

4 `Double x = 1.5; // autoboxing : double vers Double`
5 `double y = x; // unboxing : Double vers double`
6 `Double z = y; // autoboxing : double vers Double`



©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

4/40

MÉTHODE `toString()`

```
1 public class Point {  
2     private double x,y;  
3     private String nom;  
4     public Point(double x2,double y2){  
5         x=x2; y=y2;  
6     }  
7     public String toString() {  
8         return "["+x+","+y+"]";  
9     }  
10 }
```

21 `Point p=new Point(2.,3.);`
22 `System.out.println(p.toString());`

Quel est l'affichage ?
[2.0,3.0]

La méthode `toString()` est une méthode standard de Java, elle est appellée automatiquement dans certains cas.

Par exemple :

- quand on demande d'afficher un objet

23 `System.out.println(p); // Affiche : [2.0,3.0]`

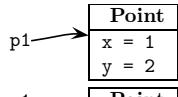
- quand on concatène une chaîne et un objet

24 `System.out.println("p=" + p); // Affiche : p=[2.0,3.0]`

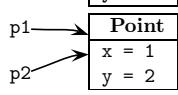
⇒ Bien souvent, il est inutile d'écrire `p.toString()`, il suffit d'écrire `p`

REPRÉSENTATION MÉMOIRE

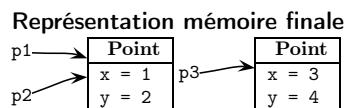
1 `Point p1 = new Point(1,2);`



2 `Point p2 = p1;`



3 `Point p3 = new Point(3,4);`



A la fin de ces 3 instructions :

- Combien d'objets créés ?
 - ◆ 2 (lignes 1 et 3 quand on fait `new Point(...)`)
- Combien de variables de type `Point` ?
 - ◆ 3 (p1, p2 et p3)
- Combien de variables d'instance x ?
 - ◆ 2 (une pour chaque objet)



©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

6/40

PROGRAMME

1 Remarques diverses

2 Notion de composition

- UML et composition

3 Programmation objet : divers

4 Javadoc, débogage : devenir autonome...

NOTION DE COMPOSITION

En POO, la composition ...

- est un type de **relation entre les classes**
- indique une relation de dépendance de type "AVOIR" entre les classes

Exemple :

- un segment **est composé** de deux points
 - ◆ reformulation : un segment a (**AVOIR**) deux points
 - ⇒ La classe Segment aura deux variables d'instance de type Point
- par contre, un point n'a pas de segments

Remarque : un type de relation similaire est l'**agrégation**. Dans l'UE LU2IN002, pour simplifier, on appellera "relation de composition", les relations de composition, mais aussi les relations d'agrégation.

COMPOSITION : SYNTAXE

```
1 public class Segment {  
2     private Point a, b; // déclaration, pas d'objets créés  
3  
4     public Segment(Point a, Point b) {  
5         this.a = a;  
6         this.b = b;  
7     }  
8     public Segment() {  
9         a=new Point(); // this(new Point(), new Point());  
10        b=new Point();  
11    }  
12    public void move(double dx, double dy) {  
13        a.move(dx, dy); // vision public du Point  
14        b.move(dx, dy);  
15    }  
16    public String toString() {  
17        return "Segment["a=" + a.toString() + ", b=" + b.toString() + "]";  
18    } // return "Segment [a=" + a + ", b=" + b + "]"; // OK  
19 }
```

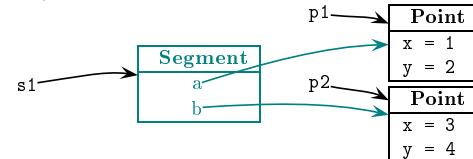
★ En général, quand il y a composition :

- ◆ la méthode `toString()` de la classe qui agrège (ici `Segment`) appelle la méthode `toString()` des variables d'instance (ici, `a` et `b`) qui sont des objets

COMPOSITION ET PRÉSENTATION MÉMOIRE

```
31 Point p1=new Point(1,2);  
32 Point p2=new Point(3,4);  
33 Segment s1=new Segment(p1,p2);
```

Représentation mémoire



★ Quelle est la représentation mémoire des instructions suivantes ?

```
41 Point p1=new Point(1,2);  
42 Segment s1=new Segment(p1,new Point(3,4));
```

- Même représentation mémoire, mais sans la variable `p2`
- `Segment s1=new Segment();`
- Même représentation mémoire, mais sans les variables `p1` et `p2`, les valeurs des `x` et des `y` sont 0

COMPOSITION

Un objet complexe = un objet qui utilise des objets

- Chaque classe reste **petite, lisible et facile à débugguer**
- Mais on peut construire des concepts complexes

```
61 Point p=new Point(1,2);  
62 Segment s1=new Segment(p,new Point());  
63 Segment s2=new Segment();
```

■ Combien d'objets de la classe `Segment` ?

⇒ 2 objets de la classe `Segment`

■ Combien d'objets de la classe `Point` ?

⇒ 4 objets de la classe `Point` (2 par objet `Segment`)

- ◆ 1 objet à la ligne 61 (`new Point(1,2)`)
- ◆ 1 objet à la ligne 62 (`new Point()`)
- ◆ 2 objets dans le constructeur sans paramètre (voir lignes 9 et 10 du slide 9)

RAPPELS : UML (UNIFIED MODELING LANGUAGE)

UML est un **langage de modélisation** standard

Dans l'UE LU2IN002, le but de ce cours n'est pas d'apprendre UML, mais d'en utiliser une **version simplifiée** comme un outils pour modéliser :

■ les **relations entre les classes** par un **diagramme de classes**

- ◆ les **classes** sont représentées par un **rectangle composé de 3 parties** :
 - 1/nom de la classe, 2/attributs, 3/méthodes
- ◆ les relations entre les classes par des **lignes spéciales** entre les rectangles

Point
- x : double
- y : double
+ Point(double,double)
+ toString() : String

■ les **objets dans la mémoire** par un **diagramme mémoire**

- ◆ les **objets** sont représentées par un **rectangle composé de 2 parties** :
 - 1/type de l'objet, 2/attributs
- ◆ les liens entre variables et objets par des **lignes fléchées**

Point
p1
x = 1
y = 2

Remarque : parfois au lieu de **diagramme**, on dit **schéma** ou **représentation**. Exemples : dessiner le schéma des classes, représenter les objets dans la mémoire

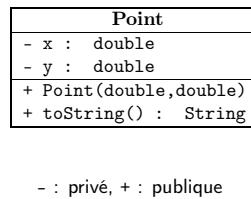
REPRÉSENTATION UML

- △ Ne pas confondre **diagramme de classe** ...
 - où on représente les classes

```

1 public class Point {
2   private double x,y;
3   private String nom;
4   public Point(double x2,double y2) {
5     x=x2; y=y2;
6   }
7   public String toString() {
8     return "["+x+","+y+"]";
9   }
10 }

```



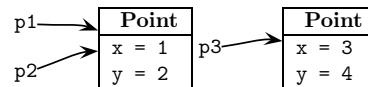
- △ ... et **diagramme mémoire**

- où on représente les objets

```

11 // main
12 Point p1 = new Point(1,2);
13 Point p2 = p1;
14 Point p3 = new Point(3,4);

```



©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 13/40

PROGRAMME

1 Remarques diverses

2 Notion de composition

3 Programmation objet : divers

- Type de base vs Objet
- Copie d'objet
- Constructeur de copie
- Égalité référentielle
- Égalité structurelle

4 Javadoc, débogage : devenir autonome...

TYPE DE BASE vs OBJET : SIGNIFICATION DE =

Affectation =

- Cas des **types de base** (int, double, char, boolean...)

```

1 int c = 1;
2 int d = c; // affectation de la valeur 1
3 d = 5;

```

⇒ Quand d est modifié, pas d'incidence sur c

- et pour un **objet** ?

```

4 Point pA = new Point(1,2);
5 Point pB = pA; // affectation de la valeur de la référence

```

⇒ 2 variables mais 1 seul objet

Si on utilise pB pour modifier l'objet :

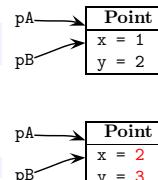
```
6 pB.move(1,1);
```

alors on peut voir que cela a modifié l'objet référencé par pA, car c'est le même objet

```

7 System.out.println(pA.toString()); // Affiche : [2,3]
8 System.out.println(pB.toString()); // Affiche : [2,3]

```



©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 17/40

UML : COMPOSITION/AGRÉGATION

- La relation de composition/agrégation est représentée par une ligne avec un losange du côté de la classe qui agrège

```

1 public class Segment {
2   private Point a,b;
3   ...

```

Diagramme de classe détaillé

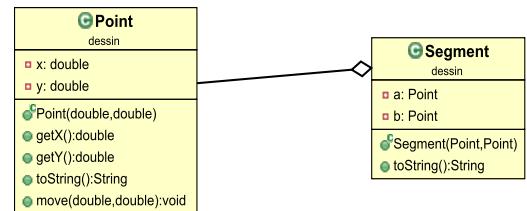
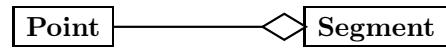


Diagramme de classe simplifié



©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 14/40

PROBLÉMATIQUE

- Le signe = se comporte de manière spécifique avec les objets...
- Le signe == également spécifique avec les objets...

Vocabulaire (uniquement pour les opérations sur objets)

new : instantiation / création d'instance

= : affectation de la valeur de la référence

== : égalité référentielle



```

1 Point p = new Point(1,2);
2 Point q = p;
3 Point r = new Point(1,2);
4 System.out.println( p == q ); // true
5 System.out.println( p == r ); // false

```

★ Comment faire pour qu'ils soient égaux si ils ont les mêmes coordonnées (**égalité structurelle**) ? Voir à partir du slide 27

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 16/40

RÉFÉRENCES ET PARAMÈTRES DE FONCTIONS

- Passer un argument à une fonction revient à utiliser un signe =
- ... objets et types de base se comportent différemment !

```

1 public class UnObjet{
2   ...
3   public void fonction1(int d){
4     ...
5     d = 3; // syntaxe correcte
6     // mais très moche !
7   }
8   public void fonction2(Point pB){
9     ...
10    // L'objet est modifié
11    pB.move(1., 1.);
12  }
13 }
21 UnObjet obj = new UnObjet();
22 ...
23 int c = 2;
24 // c vaut 2
25 obj.fonction1(c);
26 // c vaut toujours 2
31 Point pA = new Point(1,2);
32 // avant (x=1,y=2)
33 obj.fonction2(pA);
34 // après (x=2,y=3)

```

⇒ pA et pB réfèrent au même objet

- Quand un **type de base** est en argument :

◆ il y a **copie de la valeur** de la variable

- Quand un **objet** est passé en argument :

◆ **il n'y a pas copie de l'objet**

◆ mais il y a **copie de la valeur de la référence** vers l'objet

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 18/40

COPIE D'OBJETS

- ★ Comment créer une copie d'un objet?

Idée (assez raisonnable somme toute)

Créer un **nouvel objet** dont les valeurs des **attributs sont similaires**

- A si l'attribut est de **type de base**, alors **affectation de valeur**
- B si l'attribut est de **type objet**, alors il faut **copier l'objet**

Solutions classiques pour copier un objet :

- 1 constructeur de copie
- 2 méthode standard `clone()` (étudié plus tard)

Remarque : on n'a pas souvent besoin de copier un objet, par contre, les notions nécessaires à la copie d'un objet permettent de mieux comprendre la composition et la programmation objet en général

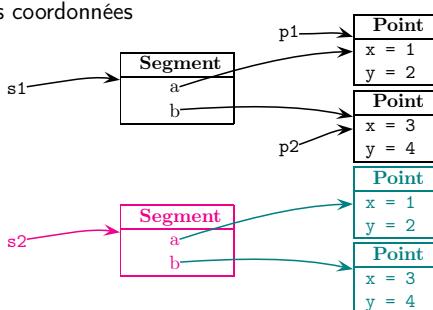
COPIE D'OBJETS : CONSTRUCTEUR DE COPIE (2/3)

- B Si l'attribut est un **objet**, il faut **copier l'objet**

♦ Exemple : constructeur de copie de la classe `Segment`

```
21 public Segment(Segment s){    31 Point p1=new Point(1,2);  
22   a = new Point(s.a); //copie    32 Point p2=new Point(3,4);  
23   b = new Point(s.b); //copie    33 Segment s1 = new Segment(p1,p2);  
24 }                                34 Segment s2 = new Segment(s1);
```

⇒ il y a 2 segments et 4 points. Chaque segment a des points de mêmes coordonnées



COPIE D'OBJETS : MÉTHODE `clone()` (1/2)

△ Cette solution sera étudiée plus tard

Solution 2 : méthode standard `clone()`

Méthode qui **retourne un nouvel objet** qui est une copie du point courant

- A Si l'attribut est de **type de base**, il suffit de faire une affectation ($\leftarrow \rightarrow$ **passage de sa valeur** en paramètre)
- Exemple de code dans la classe `Point`

```
1 public class Point {  
2   private double x, y;  
3   ...  
4   public Point clone() {  
5     return new Point(x, y); // passage des valeurs de x et y  
6   }  
7 }
```

■ Usage :

```
8 Point p1=new Point(1,2);  
9 Point p2=p1.clone();  
Remarque : construction d'un nouvel objet sans new explicite
```

COPIE D'OBJETS : CONSTRUCTEUR DE COPIE (1/3)

Solution 1 : constructeur de copie

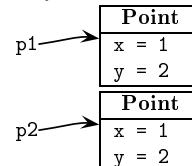
Constructeur qui prend en **paramètre un objet de même type** et qui pour chaque attribut du paramètre duplique l'attribut et l'affecte à l'attribut correspondant de l'objet courant

- A Si l'attribut est de **type de base**, il suffit d'une **affectation**

♦ Exemple : constructeur de copie de la classe `Point`

```
1 public Point(Point p) {  
2   x = p.x; // affectation  
3   y = p.y; // affectation  
4 }
```

Représentation mémoire



Copier l'objet référencé par `p1` ?

```
11 Point p1 = new Point(1,2);  
12 Point p2 = new Point(p1);
```

⇒ il y a 2 objets Point avec les mêmes valeurs d'attributs

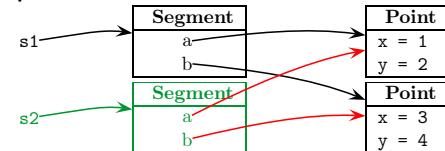
COPIE D'OBJETS : CONSTRUCTEUR DE COPIE (3/3)

★ Que se passerait-il si on ne copiait pas les points du segment ?

```
41 public Segment(Segment s){  
42   this.a=s.a; FAUX  
43   this.b=s.b; FAUX  
44 }
```

```
51 Point p1=new Point(1,2);  
52 Point p2=new Point(3,4);  
53 Segment s1=new Segment(p1,p2);  
54 Segment s2=new Segment(s1);
```

Représentation mémoire de la solution fausse



Problème : les 2 segments partagent les mêmes deux points. Si on déplace un point, les deux segments bougent !!!!

COPIE D'OBJETS : MÉTHODE `clone()` (2/2)

- B Si l'attribut est un **objet**, il faut **copier l'objet**

■ Exemple de code dans la classe `Segment`

```
1 public class Segment {  
2   private Point a, b;  
3   ...  
4   public Segment clone() {  
5     return new Segment(a.clone(), b.clone()); // copie des variables  
6   } // d'instance  
7 }
```

■ Usage :

```
8 Point p1=new Point(1,2);  
9 Point p2=new Point(3,4);  
10 Segment s1=new Segment(p1,p2);  
11 Segment s2=s1.clone();
```

Comparaison constructeur de copie et méthode `clone()`

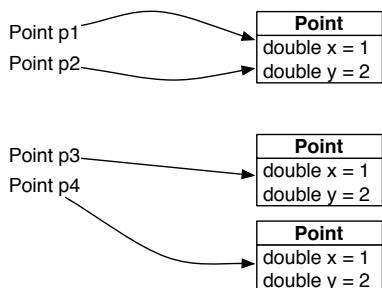
- Résultat ABSOLUMENT identique
- Cas d'utilisation un peu différent

CRÉATION DE POINTS vs AFFECTATION

```

1 Point p1 = new Point(1, 2);
2 Point p2 = p1;
3
4 Point p3 = new Point(1, 2);
5 Point p4 = new Point(1, 2);

```



- Les variables p1 et p2 référencent la même instance
- p3 et p4 réferent des instances différentes

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

25/40

COMMENT TESTER L'ÉGALITÉ STRUCTURELLE?

Idée (toujours assez raisonnable)

Créer une méthode qui teste l'égalité des variables d'instance

Solution 1 (simple mais pas utilisée)

```

1 // Dans la classe Point
2 public boolean egalite(Point p){
3     return p.x == x && p.y == y;
4 }
5
6 Point p1 = new Point(1., 2.);
7 Point p2 = p1;
8 Point p3 = new Point(1., 2.);
9 Point p4 = new Point(1., 3.);
10
11 p1.egalite(p2); // true
12 p1.egalite(p3); // true
13 p1.egalite(p4); // false

```

◆ public boolean egalite(Point p) produit le résultat attendu

◆ **ATTENTION** à la signature :

- la méthode retourne un booléen
- la méthode ne prend qu'un paramètre (on teste l'égalité entre l'instance qui invoque la méthode et l'argument)

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

27/40

MÉTHODE STANDARD : boolean equals(Object o)

Solution 2 : méthode standard equals (un peu plus complexe)

△ Cette solution sera étudiée plus tard

■ equals existe dans tous les objets (comme toString)

- par défaut : test de l'égalité référentielle...
→ pas intéressant (comme toString en version de base)

■ ⇒ Redéfinition : faire en sorte de tester les attributs

Un processus en plusieurs étapes :

- Vérifier s'il y a égalité référentielle et référence null
- Vérifier le type de l'Object o (cf cours polymorphisme)
- Convertir l'Object o dans le type de la classe (idem)
- Vérifier l'égalité entre attributs

```

1 public boolean equals(Object obj) {
2     if (this == obj) return true;
3     if (obj == null) return false;
4     if (getClass() != obj.getClass())
5         return false;
6     Point other = (Point) obj;
7     if (x != other.x) || (y != other.y)
8         return false;
9     return true;
10 }

```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

29/40

TYPE DE BASE vs OBJET : SIGNIFICATION DE ==

L'opérateur == prend 2 opérandes de **même type** et retourne un boolean

Type de base : égalité **des valeurs**

```

1 int x = 1; int y = 1;
2 System.out.println(x==y); // affichage de true

```

Type objet : égalité **valeurs des références** (égalité référentielle)

```

3 Point p1 = new Point(1, 2);
4 Point p2 = p1;
5 System.out.println(p1==p2); // affichage de true
6
7 Point p3 = new Point(1, 2);
8 Point p4 = new Point(1, 2);
9 System.out.println(p3==p4); // affichage de false

```

ATTENTION aux classes enveloppes

```

10 Integer i1 = 1 ;
11 Integer i2 = 1 ;
12 Integer i3 = new Integer(1);
13 System.out.println(i1==i2); // affichage de true
14 System.out.println(i1==i3); // affichage de false
15 System.out.println(i1.equals(i3)); // affichage de true

```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

26/40

COMMENT TESTER L'ÉGALITÉ STRUCTURELLE?

Quand il y a de la composition, il faut appeler la méthode qui compare les objets.

```

1 // Dans la classe Segment
2 public boolean egalite(Segment s){
3     return a.egalite(s.a) && b.egalite(s.b);
4 }

```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

28/40

ÉGALITÉ STRUCTURELLE : ATTENTION AU equals

△ Cette solution sera étudiée plus tard

■ Structure standard classique...

■ jusqu'au moment du test sur les attributs :

→ penser au equals (au lieu de ==)

```

1 public boolean equals(Object obj) {
2     if (this == obj) return true;
3     if (obj == null) return false;
4     if (getClass() != obj.getClass())
5         return false;
6     Voiture other = (Voiture) obj; // pour accéder aux attributs
7     if (!position.equals(other.position))
8         return false;
9     return true;
10 }

```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

30/40

PROGRAMME

- 1 Remarques diverses
- 2 Notion de composition
- 3 Programmation objet : divers
- 4 Javadoc, débogage : devenir autonome...
 - débugger son programme
 - se documenter et documenter soi-même

LES BONS REFLEXES...

- 1 Lire les messages d'erreur dans la console
- 2 Savoir corriger les erreurs les plus courantes
- 3 Savoir chercher dans la documentation officielle JAVA...
- 4 ... Et éventuellement documenter votre propre code

COMPILATEUR, JVM ET GARBAGE COLLECTOR

■ Compilateur

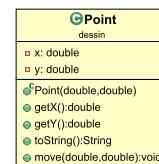
- ◆ **syntaxe** (;, parenthèses, ...)
- ◆ vérifie le **type des variables**,
- ◆ l'existence des méthodes/attributs et les niveaux d'accès :
 - les méthodes/attributs existent-elles dans l'objet,
 - les accès sont-ils permis (**public/private**)

■ JVM

- ◆ gestion dynamique des liens (cf redéfinition avec l'héritage)
- ◆ gestion des erreurs d'utilisation des objets
 - problème d'instanciation,
 - dépassement dans les tableaux,
 - gestion des fichiers...
- ◆ garbage collector (cf cycle de vie des objets)



ERREURS DE COMPILE À CORRIGER SOIT MÊME



Toujours bien regarder la ligne de l'erreur qui est donnée.
Trouver le raccourci de votre éditeur permettant d'aller à la ligne fautive

Il y a souvent plusieurs erreurs : **toujours regarder la première erreur**, les autres sont peut-être simplement des conséquences de la première

```
1 Point p = new Point(1,2);
2 p.x = 3;
3 // The field Point.x is not visible

4 p.mover(1,3);
5 // The method mover(int, int) is undefined for the type Point
6 p.move(1, 2, 3);
7 // The method move(double, double) in the type Point is
8 // not applicable for the arguments (int, int, int)

9 Point p2;
10 p2.move(1, 0);
11 // The local variable p2 may not have been initialized
```



DOCUMENTATION

Java est un langage très bien documenté et plein d'outils :

<https://docs.oracle.com/javase/8/docs/api/index.html>



ERREURS USUELLES À CORRIGER SOIT MÊME

Execution (JVM) : Toujours vérifier la ligne également

■ NullPointerException

```
1 Point p = null;
2 p.move(1, 0);
3 // Exception in thread "main" java.lang.NullPointerException
4 // at cours1.TestPoint.main(TestPoint.java:2)
```

- ◆ Cette erreur arrive souvent dans des cas plus complexes de composition d'objet

■ IndexOutOfBoundsException

```
1 int [] tab = new int [3];
2 tab[3] = 2;
3 // Exception in thread "main"
4 // java.lang.ArrayIndexOutOfBoundsException: 3
5 // at cours1.TestPoint.main(TestPoint.java:2)
```

- ◆ Vérifier la ligne et l'index !
- ◆ Souvent dans les boucles **for**



DOCUMENTER SOI-MÊME

De manière générale, on programme pour les autres...
⇒ documenter son code pour le rendre utilisable

- 1 premier niveau : choisir des noms de classes, méthodes et variables explicites.
- 2 deuxième niveau : faire des classes et des méthodes courtes, utiliser des méthodes privées...
- 3 troisième niveau : ajouter des commentaires pour créer une documentation.
 - ◆ outil intégré dans JAVA : commentaires spéciaux + création automatique d'une page web

CRÉATION D'UNE DOCUMENTATION

```
1 /**
2  * @author Vincent Guigue
3  * Cette classe permet de gérer des points en 2D
4  */
5 public class Point {
6     /**
7      * Attributs correspondant aux coordonnées du point
8      */
9     private double x, y;
10    /**
11     * Constructeur standard à partir de 2 réels
12     * @param x : abscisse du point
13     * @param y : coordonnée du point
14     */
15    public Point(double x, double y) {
16        this.x = x;
17        this.y = y;
18    }
19    /**
20     * @return l'abscisse du point
21     */
22    public double getX() {
23        return x;
24    }
25}
```

```
$ javadoc Point.java
```

JAVADOC : QUELQUES OPTIONS UTILES

- De manière générale : vérifier la documentation

```
$ javadoc -h
```

- Pour gérer les accents :

```
$ javadoc -encoding utf8 -docencoding utf8 -charset utf8 [fichier.java]
```

- Pour sélectionner le répertoire de stockage du html :

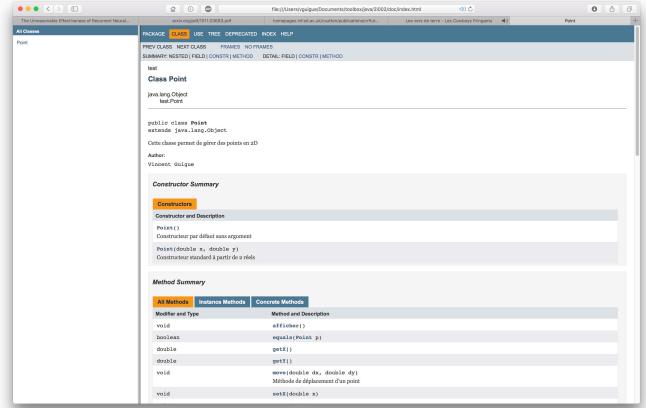
```
$ javadoc -d <directory> [fichier.java]
```

- Représentation public/private
(par défaut, représentation de la partie public seulement)

```
$ javadoc -public/-private [fichier.java]
```

JAVADOC : RÉSULTATS OBTENUS

- Classe Point, présentation conforme à la javadoc standard (présence des liens hypertextes...)



LU2IN002 - Introduction à la programmation orientée objet

Responsable de l'UE et du cours du vendredi :
Christophe Marsala
(e-mail : Christophe.Marsala@lip6.fr)

Cours du lundi : Sabrina Tollari
(e-mail : Sabrina.Tollari@lip6.fr)

(support réalisé à partir de ceux de Christophe Marsala et de Vincent Guigue)



Cours 4 – lundi 25 septembre 2023

PROGRAMME

1 Divers

- Mot clé final

2 Les tableaux

3 Classe ArrayList et fonctions avancées pour les tableaux

PROGRAMME DU JOUR

1 Divers

- Mot clé final

2 Les tableaux

- Tableau à une dimension
- Tableau du main
- Boucle for sans indice pour les tableaux
- Tableau en variable d'instance
- Tableau d'objets
- Tableau à deux dimensions
- Matrice triangulaire
- ArrayIndexOutOfBoundsException

3 Classe ArrayList et fonctions avancées pour les tableaux

- Utilisation d'un package
- La classe ArrayList
- Fonctions avancées : Collections, Arrays...

MOT CLÉ final

Une variable final ne peut pas être modifiée après initialisation.

Exemple :

```
1 final int c=10; // OK
2 c=15; // error: cannot assign a value to final variable c
```

⚠ On n'est pas obligé d'initialiser les variables final lors de la déclaration

```
11 final int d; // OK pas initialisée
12 d=12; // OK initialisation
13 d=16; // error: variable d might already have been assigned
```

Remarques :

- les attributs, les paramètres et les variables locales à une méthode peuvent être final
- on verra plus tard, qu'on peut écrire aussi des méthodes final et des classes final



©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

4/33

MOT CLÉ final : CAS DES VARIABLES D'INSTANCE

Cas particulier des variables d'instance final

Les variables d'instance final ne peuvent être initialisées que :

- lors de la déclaration
- ou dans le constructeur,
- mais pas dans une méthode

```
1 public class MaClasse {
2     private final int a = 10; // OK init déclaration
3     private final int b;
4     private final int c;
5     public MaClasse(int b) {
6         this.a = 15; // Faux a est déjà initialisée
7         this.b = b; // OK init constructeur
8     }
9     public void maMethode() {
10         this.c=24; // Faux c ne peut être initialisée
11             // dans une méthode
12     }
13 }
```

MOT CLÉ final : EXEMPLES D'UTILISATION

Exemples d'utilisation :

- un identifiant ne doit jamais être modifié
- une constante ne doit jamais être modifiée

```
1 public class Point {
2     public final String id;
3     public static final int MAX_VALUE=10;
4     private double x, y;
5     public Point(String id) {
6         this.id=id;
7         x=Math.random()*MAX_VALUE;
8         y=Math.random()*MAX_VALUE;
9     }
10 }
```

En général, les attributs doivent être déclarés private

- ★ Pourquoi ici ces attributs ont été déclarés public ?
- Car comme ils sont déclarés final, ils ne peuvent pas être modifiés ⇒ pas de problème de sécurité des données
- Le client pourra connaître la valeur, mais pas la modifier

Remarque : le mot clé static dans la déclaration de la constante sera expliquée au cours 5

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

6/33



©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

5/33

PROGRAMME

1 Divers

2 Les tableaux

- Tableau à une dimension
- Tableau du main
- Boucle `for` sans indice pour les tableaux
- Tableau en variable d'instance
- Tableau d'objets
- Tableau à deux dimensions
- Matrice triangulaire
- `ArrayIndexOutOfBoundsException`

3 Classe `ArrayList` et fonctions avancées pour les tableaux

STRUCTURE DE DONNÉES

1 Tableau à taille fixe

- + Economie mémoire
- + Rapidité d'accès
- Peu flexible (taille fixe !)

En Java :

- ◆ syntaxe assez similaire au langage C, sauf pour la réservation mémoire
- ◆ un tableau a un comportement d'objet
- ◆ tableau de type simple, mais aussi tableau d'objets

2 Tableau à taille variable

- Gourmand en mémoire
- (Un peu) moins rapide
- + Très flexible

En Java : on peut utiliser la classe `ArrayList` pour simuler des tableaux à taille variable d'objets.

⚠ Pour les types de base, il faut utiliser les classes enveloppes (`int→Integer, double→Double, ...`)

TABLEAU

tableau \simeq ensemble de variables... facilement accessibles avec une boucle

```
1 int[] tableau = new int[2];  
2 tableau[0] = 1;  
3 tableau[1] = 4;
```

■ `tableau` est une variable de type `int[]` (ie tableau d'entiers)

■ `tableau[i]` : chaque case de tableau est de type `int`

Représentation mémoire

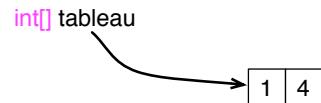


TABLEAU DU MAIN

```
1 public class TestTableauDuMain {  
2     public static void main(String[] args) {  
3         System.out.println("Il y a "+args.length+" argument(s)");  
4         for(int i=0;i<args.length;i++)  
5             System.out.println(" args["+i+"] = "+args[i]);  
6     }  
7 }
```

★ Comment utiliser le tableau du main ?

■ En ajoutant des arguments à la ligne de commande

```
> java TestTableauDuMain un deux trois  
Il y a 3 argument(s)  
args[0]=un  
args[1]=deux  
args[2]=trois  
> java TestTableauDuMain abc  
Il y a 1 argument(s)  
args[0]=abc  
> java TestTableauDuMain  
Il y a 0 argument(s)
```

SYNTAXE DES TABLEAUX

■ Déclaration d'une variable : `type[] nomVariable`

```
1 int[] tableau;
```

⚠ c'est la déclaration d'une variable, le tableau n'est pas créé

■ Instanciation : `nomVariable = new type[taille];`

```
2 tableau = new int[2];
```

⇒ Réservation de 2 cases mémoires de type `int`

■ Accès à la case `i` (lecture ou écriture) : `nomVariable[i]`

```
3 tableau[0] = 1;  
4 tableau[1] = 4;  
5  
6 int x = tableau[0];
```

■ Accès à la longueur du tableau : `nomVariable.length`

⚠ un tableau a un comportement d'objet : il a un attribut `length`

```
7 System.out.println("Longueur : "+tableau.length);
```

VARIANTES DE SYNTAXE

Pour en même temps réserver la mémoire et initialiser un tableau, on peut aussi utiliser les syntaxes suivantes.

■ Syntaxe simplifiée :

```
type[] nomVariable = {value,value,...};  
1 int[] tab1 = {1,2,3};  
2 boolean[] tab2 = {true , false , true};  
3 double[] tab3 = {1.5 , 2.3};  
4 char[] tab4 = {'a' , 'b' , 'c'} ;
```

⚠ `{value,value,...}` ne marche que lors de la déclaration

```
5 int[] tab5; // OK déclaration sans initialisation  
6 tab5={1,2,3}; // Erreur compilation, ce n'est pas une déclaration
```

■ Syntaxe intermédiaire (marche partout) :

```
type[] nomVariable;  
nomVariable = new type[] {value,value,...};  
7 int[] tab6;  
8 boolean[] tab7;  
9 tab6=new int[] {1,2,3};  
10 tab7 = new boolean[] {true , false , true};
```

TABLEAU DU MAIN

```
1 public class TestTableauDuMain {  
2     public static void main(String[] args) {  
3         System.out.println("Il y a "+args.length+" argument(s)");  
4         for(int i=0;i<args.length;i++)  
5             System.out.println(" args["+i+"] = "+args[i]);  
6     }  
7 }
```

★ Comment utiliser le tableau du main ?

■ En ajoutant des arguments à la ligne de commande

```
> java TestTableauDuMain un deux trois  
Il y a 3 argument(s)  
args[0]=un  
args[1]=deux  
args[2]=trois  
> java TestTableauDuMain abc  
Il y a 1 argument(s)  
args[0]=abc  
> java TestTableauDuMain  
Il y a 0 argument(s)
```

TABLEAUX ET BOUCLES

Code robuste = pas de duplication de l'information

Attention aux conditions de fin de boucles

```
1 int [] tab = {2, 3, 4, 5, 6};  
Besoins de faire une boucle...  
2 for (int i=0 ; i < 5 ; i++) // INCORRECT dans le cadre de LU2IN002  
3 ...  
4 for (int i=0 ; i < tab.length ; i++) // CORRECT  
5 ...
```

Bonne pratique de programmation

A chaque fois que c'est possible, utiliser `tab.length` pour indiquer la taille du tableau

TABLEAUX ET BOUCLES

Pour les tableaux, il existe aussi une boucle `for` sans indice

```
for(type var : nomTableau) {  
    ...  
}
```

`var` prend successivement toutes les valeurs des éléments du tableau

```
boolean [] tableau={true, false, true};  
for (boolean b : tableau) {  
    System.out.println(b); // affiche chaque case du tableau  
}
```

⚠ Cette boucle ne peut pas être utilisée dans certains cas. Exemples :

- quand on a besoin des indices
- quand on veut modifier le tableau (ici `b` est une variable locale)

Remarque : cette boucle `for` sans indice peut aussi être utilisée :

- avec les tableaux d'objets
- avec certaines classes, dont la classe `ArrayList`

TABLEAUX : REMARQUES

■ Un tableau a un comportement d'objet

```
1 int [] tab1 = {1,2,3};  
2 int [] tab2 = tab1;  
3 int [] tab3 = {1,2,3};  
4  
5 System.out.println(tab1==tab2); // Affichage : true  
6 System.out.println(tab1==tab3); // Affichage : false
```

■ Un tableau peut ne contenir aucune case

```
7 int [] tabVide=new int [0];  
8 System.out.println("Taille = "+tabVide.length);  
9 // Affichage : Taille = 0
```

TABLEAU EN VARIABLE D'INSTANCE (2/3)

⚠ Quand il y a plusieurs constructeurs, il faut faire attention que, dans tous les cas, la réservation mémoire pour le tableau soit effectuée

```
1 public class MaClasse {  
2     private int [] tab;  
3     public static final int TAILLE_STANDARD=10;  
4     public MaClasse(int taille) {  
5         tab=new int [taille]; // réserve la mémoire  
6     }  
7     public MaClasse() {  
8         this(TAILLE_STANDARD); // réserve la mémoire grâce à ligne 5  
9     }  
10    public MaClasse(int x, int y) {  
11        tab[0]=x; tab[1]=y; // ERREUR le tableau n'a pas été créé  
12    }
```

Correction de l'erreur :

```
13    public MaClasse(int x, int y) {  
14        tab=new int[2]; //OU this(2); (réserve la mémoire ligne 5)  
15        tab[0]=x; tab[1]=y;  
16    }
```

TABLEAU EN VARIABLE D'INSTANCE (1/3)

★ Comment utiliser un tableau en variable d'instance ?

```
1 public class MaClasse {  
2     private int [] tab; // déclaration d'une variable  
3  
4     public MaClasse(int taille) {  
5         tab=new int [taille]; // réservation mémoire  
6     }
```

En général, quand on utilise un tableau en variable d'instance, il faut penser à **réserver la mémoire dans le constructeur**

- ⚠ Le paramètre `taille` du constructeur n'est pas du même type que l'attribut `tab`, mais il sert à initialiser l'attribut
- ⚠ Ne pas déclarer un attribut `taille`, utiliser `tab.length`

```
7 ...  
8     public String toString() {  
9         String s="[";  
10        for(int i=0 ; i < tab.length ; i++) {  
11            s = s + tab[i] + " ";  
12        }  
13        return s + "]";  
14    }
```

TABLEAU EN VARIABLE D'INSTANCE (3/3)

★ Pourquoi en général ne faut-il pas initialiser une variable d'instance avec un paramètre de type tableau ?

```
21 int [] tabMain={1,2,3};  
22 MaClasse mc=new MaClasse(tabMain);  
23 System.out.println(mc.toString());  
24 // Affiche [ 1 2 3 ]  
25 tabMain[0]=10;  
26 System.out.println(mc.toString());  
27 // Affiche [ 10 2 3 ]
```

★ Pourquoi en général ne faut-il pas faire d'accesseur pour une variable d'instance de type tableau ?

```
6 ...  
7     public int [] getTab() {  
8         return tab; // A EVITER  
9     }  
10 }
```

Solution ligne 4 : affecter non pas le paramètre, mais une copie du paramètre
Solution ligne 8 : retourner une copie du tableau de la variable d'instance

Voir le slide 33 pour copier un tableau avec la classe `Arrays`

TABLEAU D'OBJETS

Soit la classe Point (vue dans les cours précédents).

On veut faire un tableau d'objets Point.

- Déclaration d'une variable `tabP` de type `Point []`

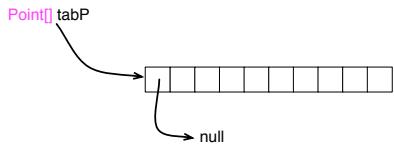
```
Point [] tabP;
```

C'est juste la déclaration d'une variable, le tableau n'existe pas encore (il n'est pas instancié)

- Instantiation du tableau (réservation des cases mémoires)

```
tabP = new Point [10];
```

→ La variable `tabP` référence un tableau de 10 cases



10 cases = 10 variables... mais aucun objet Point

TABLEAU D'OBJETS

Les cases se comportent vraiment comme des variables : on peut jouer avec les références

```
1 Point p = new Point(3,4);
2 Point [] tabP2 = new Point [3];
3
4 tabP2[0] = p;
5 tabP2[1] = new Point(4,5);
6 tabP2[2] = tabP2[0];
```

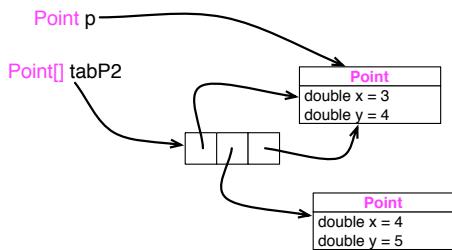


TABLEAU D'OBJETS À DEUX DIMENSIONS (1/2)

```
1 Point [][] matP = new Point [2][3];
```

est équivalent à la réservation d'un tableau de tableaux de Point

```
2 Point [][] matP2 = new Point [2][];
3 for (int i=0; i<matP2.length; i++)
4     matP2[i] = new Point [3];
```

Création des objets Point (similaire pour `matP` et `matP2`)

```
5 for (int i=0; i<matP.length; i++)
6     for (int j=0; j<matP[i].length; j++)
7         matP[i][j] = new Point();
```

Affichage avec des boucles sans indice

```
8 for (Point [] ligne : matP) {
9     for (Point p : ligne) {
10        System.out.println(p);
11    }
12    System.out.println();
13 }
```

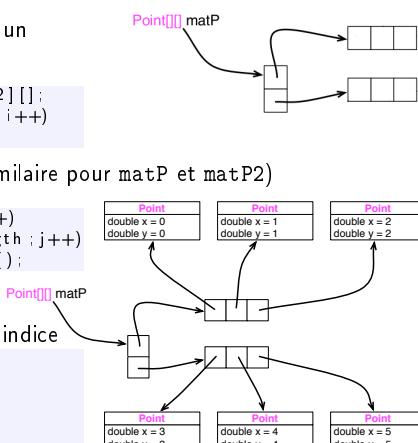


TABLEAU D'OBJETS

Chaque case (=variable) peut/doit être initialisée avec un objet

- Initialisation d'une case

```
tabP[0] = new Point(1,2);
```

- Initialisation avec une boucle

```
for (int i=1; i<tabP.length; i++)
    tabP[i] = new Point(i, i);
```

- Affichage
(boucle sans indice)

```
for (Point pi : tabP)
    System.out.println(pi);
// <=>
for (int i=0; i<tabP.length; i++) {
    Point pi=tabP[i];
    System.out.println(pi);
}
```

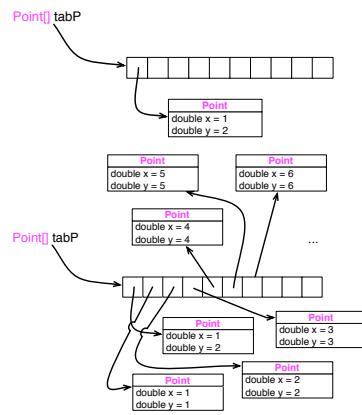


TABLEAU À DEUX DIMENSIONS

Comment gérer une matrice ? Comme un tableau de tableaux

- Déclaration des variables : `type[][]`

```
1 int [][] matrice;
```

- Instantiation

```
2 matrice = new int [2][3]; // 2 lignes, 3 colonnes
```

- Usage

```
3 matrice[0][0] = 0; matrice[0][1] = 1; matrice[0][2] = 2;
4 matrice[1][0] = 3; matrice[1][1] = 4; matrice[1][2] = 5;
```

- Syntaxe alternative d'instanciation/initialisation

```
5 int [][] matrice = {{0, 1, 2}, {3, 4, 5}};
```

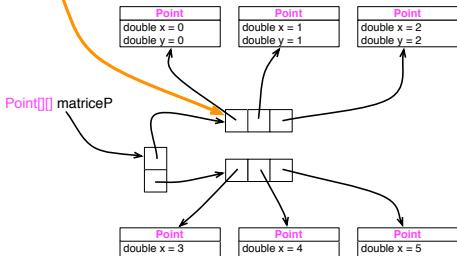
- Accès aux dimensions

```
6 matrice.length // nb lignes
7 matrice[0].length // nb de colonnes de la première ligne
8 matrice[1].length // nb de colonnes de la deuxième ligne
```

- `matrice` est de type `int[][]`, `matrice[0]` est de type `int[]`, `matrice[0][0]` est de type `int`

TABLEAU D'OBJETS À DEUX DIMENSIONS (2/2)

Point[] ligne



- Possibilité de manipuler les lignes de la matrice de manière indépendante

```
1 Point [][] matriceP = new Point [2][3];
```

```
2 Point [] ligne = matriceP[0];
```

```
3 // Affichage du premier point :
```

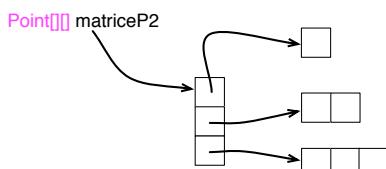
```
4 System.out.println(ligne[0]); // <=> matriceP[0][0]
```

MATRICE TRIANGULAIRE

Pour éviter de réservé inutilement de la mémoire, on peut créer des tableaux de tableaux où les lignes ont des tailles différentes

- La déclaration s'effectue généralement en 2 étapes :
 - 1 d'abord, on déclare un tableau de tableaux **sans préciser la deuxième dimension**
 - 2 puis, pour chaque ligne, on déclare un tableau à la bonne taille
- Par exemple, une matrice triangulaire

```
1 Point[][] matriceP2 = new Point[3][]; // Etape 1
2 for(int i=0; i<matriceP2.length; i++)
3     matriceP2[i] = new Point[i+1]; // Etape 2
```



Rappel : aucun objet Point n'a été créé

DÉPASSEMENT DE TABLEAU

⚠ Tableau... ⇒ possibilité de dépasser dans un tableau

- Cas classique :
 - ◆ Mélange entre taille n et dernier indice du tableau ($n - 1$)
 - ◆ Tentative d'accès à un index négatif
 - ◆ Erreur de boucle...
- Symptôme : `ArrayIndexOutOfBoundsException`
 - ◆ Echec lors de l'exécution du code (compilation OK)

```
1 Point[] tab = {new Point(), new Point()};
2 System.out.println(tab[2]); // pas de troisième case
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
at test.Point.main(Point.java:118)
```

■ Attention aux `NullPointerException` : après instantiation d'un tableau, aucune instance n'est disponible :

```
3 Point[] tab = new Point[2];
4 System.out.println(tab[0].getX()); // => NullPointerException
```

PROGRAMME

1 Divers

2 Les tableaux

3 Classe ArrayList et fonctions avancées pour les tableaux

- Utilisation d'un package
- La classe `ArrayList`
- Fonctions avancées : `Collections`, `Arrays`...

LA CLASSE ArrayList

Usage dans 2 cas (imbriqués) :

- Taille finale inconnue lorsque l'on commence à utiliser le tableau (e.g. lecture d'un fichier...)
- Taille variable en cours d'utilisation (e.g. pile d'objets à traiter de taille variable)
- Syntaxe objet classique + approche générique (hors prog.) :
 - ◆ la variable sera de type : `ArrayList<Type>`
 - ◆ `Type` est forcément un type objet
 - ⇒ Pour les types de base, il faut utiliser les classes enveloppes (`Integer`, `Double`, ...)
 - ◆ exemples : `ArrayList<Point>`, `ArrayList<Integer>`
- Même représentation mémoire que les tableaux de taille fixe

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 25/33

LES PACKAGES

- Java est fourni avec un ensemble de classes déjà programmées
- Ces classes sont regroupées en fonction de leurs fonctionnalités dans des ensembles de classes appelés **package**

`package` ≈ bibliothèque de classes

Au démarrage, Java importe automatiquement le package `java.lang` qui contient notamment les classes :

- `String`, `Math`, `System`...

Pour utiliser une classe d'un autre package, il faut importer la classe. Par exemple:

- la classe `ArrayList` se trouve dans le package `java.util`
- pour utiliser la classe `ArrayList`, il faut écrire **au début de chaque fichier qui utilise cette classe** :

```
import java.util.ArrayList;
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 28/33

ArrayList : SYNTAXE DÉTAILLÉE

- Déclaration et création d'un objet de type `ArrayList<Point>`

```
1 ArrayList<Point> alp = new ArrayList<Point>();
2 alp.add(new Point(1,2));
3 for(int i=0; i<9; i++)
4     alp.add(new Point(i,i));
```

- Accès aux éléments

```
5 // Pour obtenir une référence sur le 1er élément,
6 // mais sans le supprimer de la liste
7 Point p1 = alp.get(0);
8 // Pour obtenir une référence sur le 1er élément
9 // ET le supprimer de la liste
10 Point p2 = alp.remove(0);
```

- Nombre d'éléments

```
11 System.out.println(alp.size());
```

Plus d'informations dans la javadoc (beaucoup d'autres méthodes disponibles) :

<https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html>

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 29/33

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

30/33

ArrayList : SYNTAXE DÉTAILLÉE

⚠ Supprimer un élément avec `remove`, entraîne un décalage

```
1 ArrayList<Double> al = new ArrayList<Double>();
2 al.add(11.); al.add(12.); al.add(13.);
3
4 System.out.println("Avant le remove size=" + al.size());
5 for (int index=0; index<al.size(); index++)
6     System.out.println("Element " + index + ":" + al.get(index));
7
8 Double x=al.remove(1); // On retire l'élément en 2ième position
9 System.out.println("x=" + x);
10
11 System.out.println("Après le remove size=" + al.size());
12 for (int index=0; index<al.size(); index++)
13     System.out.println("Element " + index + ":" + al.get(index));

Avant le remove size=3
Element 0: 11.0
Element 1: 12.0
Element 2: 13.0
x=12.0
Après le remove size=2
Element 0: 11.0
Element 1: 13.0 // décalage : le 3ème élément se retrouve en 2ième position
```

FONCTIONS AVANCÉES...

Java est fourni avec un grand nombre de classes avec des fonctions utilitaires déjà programmées pour les tableaux, les `ArrayList` et autres classes qui gèrent un ensemble d'objets

A) ... pour les `ArrayList`

```
1 ArrayList<Integer> ali = new ArrayList<Integer>();
2 for (int i = 0; i < 10; i++)
3     ali.add((int)(Math.random()*10));
```

■ Dans la classe `ArrayList`:

```
4 if (ali.contains(2))
5     System.out.println("Valeur trouvée !");
6 for (Integer x : ali) // boucle for sans indice
7     System.out.println(x);
```

■ Dans la classe `Collections` du package `java.util`

◆ Tris, min, max, mélange, renversement...

```
8 System.out.println(ali);
9 int min=Collections.min(ali);
10 System.out.println("Min=" + min);
11 Collections.sort(ali);
12 System.out.println(ali);
13 Collections.reverse(ali);
14 System.out.println(ali);

Affichage :
[2, 3, 5, 9, 3, 6, 1, 3, 8, 3]
Min=1
// tri
[1, 2, 3, 3, 3, 5, 6, 8, 9]
// inversion
[9, 8, 6, 5, 3, 3, 3, 3, 2, 1]
```

FONCTIONS AVANCÉES...

B) ... pour les tableaux

■ La classe `Arrays` du package `java.util`

◆ recherche, affichage, tri, copie, remplissage...

■ Exemple :

```
1 import java.util.Arrays;
2
3 // main
4 int [] tab={13,15,11,14,12};
5 int index=Arrays.binarySearch(tab, 14); // recherche l'index de 14
6 System.out.println("Index=" + index);
7 System.out.println("tab=" + Arrays.toString(tab)); // affichage
8 Arrays.sort(tab); // tri du tableau
9 System.out.println("tab=" + Arrays.toString(tab));
9 int [] tabCopie=Arrays.copyOf(tab, tab.length); // copie
```

■ Affichage :

```
Index=3
tab=[13, 15, 11, 14, 12]
tab=[11, 12, 13, 14, 15]
```

LU2IN002 - Introduction à la programmation orientée objet

Responsable de l'UE et du cours du vendredi :
Christophe Marsala
(e-mail : Christophe.Marsala@lip6.fr)

Cours du lundi : Sabrina Tollari
(e-mail : Sabrina.Tollari@lip6.fr)

(support réalisé à partir de ceux de Christophe Marsala et de Vincent Guigue)



Cours 5 – lundi 2 octobre 2023

PROGRAMME

1 Static

- Usage et syntaxe
- Les constantes
- Classe "Outil"
- Utilisation dans une autre classe

2 Combiner static et POO : exemples

3 Héritage

PROGRAMME DU JOUR

1 Static

- Usage et syntaxe
- Les constantes
- Classe "Outil"
- Utilisation dans une autre classe

2 Combiner static et POO : exemples

- Compteur d'instances
- Génération automatique d'identifiant
- Garder une liste des objets créés
- Singleton

3 Héritage

- Rappels des principes de la POO
- Premières notions sur l'héritage

POO ≠ STATIC

POO

- Un objet protège ses attributs
- Un objet possède des méthodes pour gérer ses attributs

Static

- Les attributs/méthodes static ne dépendent pas d'un objet
- Tous les objets d'une classe ont accès aux mêmes informations static

Usage

- 1 Création d'un objet
- 2 Appel de méthodes sur cet objet

Usage

- 1 Appel de méthode/attribut indépendamment des objets

Certains problèmes sont par nature des problèmes plutôt orientés objets, tandis que d'autre non

- Par exemple, pour générer un nombre aléatoire, inutile de créer un objet

EXEMPLES DE CAS D'USAGE DE STATIC

■ Attribut static : partage d'information entre objets de la classe

- ◆ Constantes : TAILLE_MAX, π ...
- ◆ Compteurs

Combien d'instances de Point ont-elles été créées?
Question non triviale avec les outils actuels !

- ◆ Liste des objets créés

Je voudrais accéder à n'importe quel point créé jusqu'ici...

■ Méthode static : méthode non liée à un objet

- ◆ méthode "outil"

Calculer le cosinus est un problème qui ne dépend pas d'un objet

- ◆ accesseur d'un attribut static
- ◆ méthode main
- ◆ l'exemple du Singleton



©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

4/26

SYNTAXE : VARIABLE STATIC

Deux sortes d'attributs :

- variable d'instance
- variable de classe (variable static)

```
1 public class MaClasse {  
2     private int var1;  
3     private static int varC = 0;  
4  
5     public MaClasse(int var1) {  
6         this.var1=var1;  
7         varC++;  
8     }  
}
```

Bonne pratique

En général :

- les variables d'instance s'initialisent dans le constructeur
- les variables de classe s'initialisent lors de la déclaration

Remarque : comme les variables static sont déjà initialisées avant la création du premier objet, on peut les utiliser dans le constructeur

SYNTAXE : MÉTHODE STATIC

```
1 public class MaClasse {  
2     private int var1;  
3     private static int varC = 0;  
4  
5     public MaClasse(int var1) {  
6         this.var1=var1;  
7         varC++;  
8     }  
9     public static int methodeStatic(int a) {  
10        // instructions qui ne dépendent  
11        // pas d'un objet. Exemple :  
12        return a+varC;  
13    }  
14    public static int getVarC() {  
15        return varC;  
16    }  
17 }
```

Deux sortes de méthodes :

- méthode d'instance
- méthode de classe (méthode static)

Bonne pratique

En général, l'accesseur d'une variable static doit être static, car il ne dépend pas d'un objet

SYNTAXE : CLASSE "OUTIL"

Pour certains problèmes, on n'a pas besoin d'objets, mais d'une classe pour "stocker" des valeurs et faire des "calculs"

- Par exemple, la classe Math contient la variable PI pour "stocker" π et des méthodes pour calculer cos, sin, ...
- De même, on peut aussi écrire une classe qui contient seulement :
 - ◆ des attributs static
 - ◆ des méthodes static
 - ◆ un constructeur privé pour empêcher la création d'objets

```
1 public class MaClasseOutil {  
2     public static final int MA_CONSTANTE=10;  
3     private static int autreAttributStatic=15;  
4     private MaClasseOutil() { }  
5     public static void maMethode() { }  
6 }
```

△ Comme le constructeur est privé, on ne peut pas créer d'objets dans une autre classe

// dans le main d'une classe Test
MaClasseOutil mco = new MaClasseOutil(); Erreur compilation

SYNTAXE/PHILOSOPHIE COMPARATIVE

Programmation objet

```
1 // Instantiation  
2 Point p = new Point(1,2);  
3  
4 // Invocation de méthode  
5 // sur l'objet  
6 p.move(3, 3);  
7 ...
```

Philosophie

Les méthodes accèdent / modifient l'objet

⇒ Essayons maintenant de mélanger les 2 philosophies pour faire des choses nouvelles

Programmation static

```
1 // Pas d'instantiation de la classe  
2 // Appel directement sur la classe  
3 double pi = Math.PI;  
4  
5 // Pareil pour les méthodes  
6 double d = Math.cos(pi);
```

Philosophie

- Pas d'instance, pas d'accès aux variables d'instance
- Constante indépendante
- Méthode indépendante

SYNTAXE : LES CONSTANTES

```
1 public class MaClasse {  
2     public static final int MA_CONSTANTE=10;  
3     ...  
4     ■ public : si le client est autorisé à connaître la valeur de la constante, private sinon  
5     ■ static : une constante ne dépend pas d'un objet  
6     ■ final : une constante ne doit pas être modifiée
```

△ Ne pas confondre les mots clés static et final

Cas particulier des tableaux

```
11 public class MaClasse {  
12     private static final int [] tab={11,12,13};  
13     public static void maMethode() {  
14         tab[0] = 15; // OK modification de la valeur d'une case  
15         tab = new int [4]; // Faux, la variable tab est final  
16     }  
17 }
```

Pour les tableaux (et les objets), c'est la valeur de la variable qui ne peut pas changer, par contre, les valeurs des cases du tableau (et les objets) peuvent changer ⇒ En général, déclarer la variable private

SYNTAXE : UTILISATION DANS UNE AUTRE CLASSE

```
1 public class MaClasseOutil {  
2     public static final int MA_CONSTANTE=10;  
3     private MaClasseOutil() { }  
4     public static void maMethode() { }  
5 }
```

★ Comment utiliser une variable static ou une méthode static dans une autre classe ? Par exemple dans le main de Test ?

Variable static : NomClasse.nomVariableStatic

Exemples :
21 System.out.println(MaClasseOutil.MA_CONSTANTE);
22 double x = Math.PI;
23 System.out.println(x); // out : variable static

Appel d'une méthode static : NomClasse.nomMethodeStatic(...)

Exemples :
24 MaClasseOutil.maMethode();
25 double y = Math.random();
26 String s = String.format("%.2f",y);

Remarque : aucune instantiation d'objet n'a été nécessaire pour utiliser ces variables et méthodes static

PROGRAMME

1 Static

2 Combiner static et POO : exemples

- Compteur d'instances
- Génération automatique d'identifiant
- Garder une liste des objets créés
- Singleton

3 Héritage

CAS CLASSIQUE : COMPTAGE D'INSTANCES

Combien d'instances de Point ont-elles été créées?
Question non triviale avec les outils actuels !

```
1 Point p1 = new Point();
2 Point p2 = p1;
3 Point p3 = new Point(3,5);
4 ...
```

- Peut-on avoir un **compteur d'instances** qui compte le nombre d'objets Point créés ?
- Peut-on attribuer à chaque Point un **identifiant** unique lié à son ordre de création ?

COMPTAGE D'INSTANCES : SYNTAXE STANDARD

```
1 public class Point {
2     private static int cpt = 0; // compteur d'instances
3     private final int id; // identifiant d'une instance
4     private double x,y;
5     public Point(double x, double y) {
6         this.x = x;
7         this.y = y;
8         cpt++;
9         id = cpt;
10    }
11 }
```

- Chaque objet de type Point possède :

- ◆ un x, un y, un **id**
- Tous les objets de type Point partagent :
- ◆ un compteur **cpt** défini au niveau de la classe

Remarque : cette classe contient la **forme standard** pour déclarer un **compteur d'instances** et l'utiliser pour donner un identifiant

- 1 déclaration d'une variable static initialisée à 0 (ligne 2)
- 2 incrémentation de la variable static dans le constructeur (ligne 8)
- 3 utilisation de la variable static pour initialiser l'identifiant id (ligne 9)

COMPTAGE D'INSTANCES : SYNTAXE STANDARD (2)

△ Piège : attention aux constructeurs multiples

```
1 public class Point {
2     private static int cpt = 0;
3     private final int id;
4     private double x,y;
5     public Point(double x, double y) {
6         this.x = x;
7         this.y = y;
8         cpt++;
9         id = cpt;
10    }
11    public Point(double d) {
12        this(d,d);
13        cpt++; // FAUX pourquoi ? Car déjà fait par le this(...) ligne précédente
14    }
15    public Point() {
16        this(Math.random()*10, Math.random()*10);
17    }
18 }
```

- Usage de **this(...)** très fortement conseillé pour toujours passer par le constructeur de référence et bien compter
- ⇒ Dans chaque constructeur, les instructions **cpt++**; et **id = cpt;** sont bien réalisées

STATIC / NON STATIC : ASYMÉTRIE

- Les instances voient ce qui est **static**
- Les parties **static** ne voient pas les instances

```
1 public class Point{
2     private static int cpt = 0;
3     private final int id;
4     ...
5     // Cas 1 : OK méthode static, accès variable static
6     public static int getCpt() { return cpt; }
7
8     // Cas 2 : OK méthode d'instance, accès variable static
9     public void methInst() {
10         System.out.println(cpt);
11     }
12     // Cas 3 : KO méthode static, pas accès variable d'instance
13     public static void methStatic() {
14         System.out.println(id); // Erreur les méthodes static ne peuvent pas
15         // utiliser de variables d'instance
16     }
17 }
```

Dans le main : **Point p1 = new Point();**

```
21 Point.getCpt(); // OK syntaxe naturelle appel méthode static
22 p1.getCpt(); // OK, mais à éviter
```

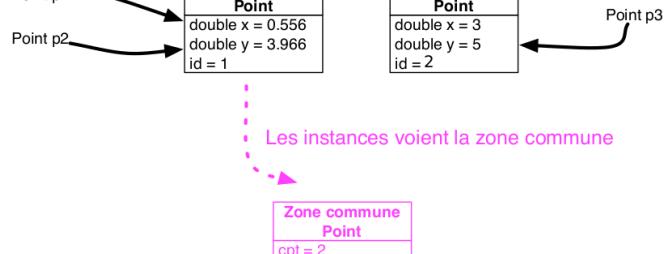
```
23 Point.methInst(); // Erreur compil : méthode d'instance nécessite un objet
24 p1.methInst(); // OK syntaxe naturelle appel méthode d'instance
```

COMPTAGE & PRÉSENTATION MÉMOIRE

```
1 public class Point {
2     private static int cpt=0; 11 Point p1 = new Point();
3     private final int id; 12 Point p2 = p1;
4     private double x,y; 13 Point p3 = new Point(3,5);
5     ...
6 }
```

- Dans la représentation mémoire :

◆ où se trouve l'**id**? où se trouve le compteur **cpt**?



△ Par contre, la zone commune ne connaît pas les instances

LISTE DES OBJETS CRÉÉS

- ★ Peut-on garder une liste des objets créés?

```
1 import java.util.ArrayList;
2 public class Point {
3     private double x,y;
4     private static ArrayList<Point> alp=new ArrayList<Point>();
5
6     public Point(double x, double y){
7         this.x = x; this.y = y;
8         alp.add(this); // ajout du point créé dans la liste
9     }
10    public String toString() {
11        return "["+x+","+y+"]";
12    }
13    public static void afficherListePoints() {
14        for(Point p : alp) {
15            System.out.println(p);
16        }
17    }
18 }
```

31 // Main
32 new Point(3,4);
33 new Point(5,6);
34 Point.afficherListePoints();

Affiche les deux points :
[3.0 ,4.0]
[5.0 ,6.0]

L'EXEMPLE DU SINGLETON

- ★ Comment garantir qu'une classe ne puisse n'avoir **qu'une seule instance** ?

Approche du patron de conception Singleton :

```
1 public class Singleton {  
2     private static final Singleton INSTANCE = new Singleton();  
3  
4     private Singleton() {}  
5  
6     public static Singleton getInstance() {  
7         return INSTANCE;  
8     }  
9 }
```

- 1 une variable **static** pour stocker la seule instance
- 2 un seul constructeur **private** pour empêcher la création d'autres instances
- 3 une méthode **static** pour obtenir la référence vers cette unique instance

Remarque : il existe plusieurs variantes de ce patron

UN EXEMPLE D'UTILISATION DU SINGLETON (2/2)

- Une classe pour représenter l'origine d'un repère orthonormé

```
1 Origine orig = Origine.getInstance();  
2 System.out.println(orig);  
3  
4 Point p1 = new Point(3, 2);  
5 System.out.println(p1);  
6  
7 double d = orig.distanceAOrigine(p1);  
8  
9 System.out.println("Distance entre "+orig+" et "+p1+" : "+d);
```

- Résultat :

```
origine (0.0, 0.0)  
(3.0, 2.0)  
Distance entre origine (0.0, 0.0) et (3.0, 2.0) : 3.605551275
```

PROGRAMME

- 1 Static
- 2 Combiner static et POO : exemples
- 3 Héritage
 - Rappels des principes de la POO
 - Premières notions sur l'héritage

UN EXEMPLE D'UTILISATION DU SINGLETON (1/2)

- Une classe pour représenter l'origine du repère orthonormé :
 - ◆ l'origine est un point unique

```
1 public class Origine {  
2     private static final Origine INSTANCE = new Origine(0,0);  
3     private double x, y;  
4  
5     private Origine(double x, double y) {  
6         this.x = x; this.y = y;  
7     }  
8     public static Origine getInstance() {  
9         return INSTANCE;  
10    }  
11    public String toString() {  
12        return "origine(" + this.x + ", " + this.y + ")";  
13    }  
14    public double distanceAOrigine(Point p) {  
15        return Math.sqrt(p.getX() * p.getX()  
16                            + p.getY() * p.getY());  
17    }  
18 }
```

BILAN...

- ★ Comment savoir si une variable est static ou pas ?

- Est-ce que la **variable dépend d'un objet** ?
 - ◆ Si oui, variable d'instance (VI)
 - ◆ Si non, variable de classe (static) (VC)

Exemple : On veut écrire une classe Personne

- age ? VI
l'âge d'une personne dépend de la personne
- AGE_MAJORITE ? VC
l'âge de la majorité ne dépend pas d'une personne en particulier
- cptPersonnes ? VC
le nombre de personnes créées ne dépend pas d'une personne en particulier
- cptEnfants ? VI **⚠️ Tous les compteurs ne sont pas static**
le nombre d'enfants d'une personne dépend de la personne

Quand on vous parle de **static**, n'oubliez pas :

- Ce sont des cas très particuliers et **assez rare**
- N'oubliez pas les bonnes pratiques de la POO!!!!!!

PRINCIPES ORIENTÉS OBJETS

Principe 1 : Encapsulation

- Rapprochement données (attributs) et traitements (méthodes)
- Protection de l'information (private/public)

Principe 2 : Composition/Agrégation

- Un objet de la classe A est composé d'objets de la classe B
- Classe A AVOIR des Classes B

Principe 3 : Héritage

- Un objet de la classe B est un objet de la classe A aussi
- Classe B ETRE une Classe A

⇒ La classe B hérite de la classe A

HÉRITAGE

Idée de l'héritage

- Spécialiser une classe : ajouter des fonctionnalités à une classe
 - ◆ Exemple : un point qui a en plus un nom
- Hériter du comportement d'une classe existante
 - ◆ Exemple : un point qui a un nom est quand même un point
- Une classe \Rightarrow plusieurs spécialisations possibles
 - ◆ Animal \rightarrow Vache, Chien, Panda...
 - ◆ hiérarchisation possible : Animal \rightarrow Insecte \rightarrow Papillon

Objectifs :

- Ne pas avoir à modifier le code existant
 - ◆ ne pas modifier la classe de base
 - ◆ Point \rightarrow PointNommé : un point avec un nom
- Ne pas avoir à faire de copier-coller!
 - ◆ faire hériter le comportement d'une classe

EXEMPLES & CONTRE EXEMPLES

Pour les cas suivants, dire si les relations sont des relations de type **Composition** ou **Héritage** :

- Salle de bains et baignoire
 - Piano et pianiste
 - Personne, enseignant et étudiant
 - Animal, chien et labrador
 - Cercle et ellipse
 - ◆ Un cercle est une ellipse particulière
 - ◆ OU Une ellipse est un cercle avec un attribut en plus
 - Entier et réel
 - ◆ Un entier est un réel particulier
 - ◆ OU Un réel est un entier avec une partie décimale en plus
- △ Pour un même problème plusieurs modélisations sont souvent possibles

LU2IN002 - Introduction à la programmation orientée objet

Responsable de l'UE et du cours du vendredi :
Christophe Marsala
(e-mail : Christophe.Marsala@lip6.fr)

Cours du lundi : Sabrina Tollari
(e-mail : Sabrina.Tollari@lip6.fr)

(support réalisé à partir de ceux de Christophe Marsala et de Vincent Guigue)



Cours 6 – lundi 9 octobre 2023

PROGRAMME

1 Héritage : syntaxe

- Rappels des principes de la POO
- Mots clefs extends et super
- Représentation mémoire et héritage
- Diagramme de classes UML et héritage
- Niveau d'accès : protected

2 Héritage : subsomption et polymorphisme

3 Héritage : surcharge / redéfinition

PROGRAMME DU JOUR

1 Héritage : syntaxe

- Rappels des principes de la POO
- Mots clefs extends et super
- Représentation mémoire et héritage
- Diagramme de classes UML et héritage
- Niveau d'accès : protected

2 Héritage : subsomption et polymorphisme

- Principe de subsomption
- Polymorphisme : application aux tableaux
- La classe Object

3 Héritage : surcharge / redéfinition

- Surcharge et héritage
- Redéfinition
- Redéfinition et usage de super
- Ouverture de la redéfinition

PRINCIPES ORIENTÉS OBJETS

Principe 1 : Encapsulation

- Rapprochement données (attributs) et traitements (méthodes)
- Protection de l'information (private/public)

Principe 2 : Composition/Agrégation

- Un objet de la classe A est composé d'objets de la classe B
- Classe A AVOIR des Classe B

Principe 3 : Héritage

- Un objet de la classe B est un objet de la classe A aussi
- Classe B ETRE une Classe A

⇒ La classe B hérite de la classe A

EXEMPLE : POINT ET POINTNOMMÉ

Problème

On a déjà écrit une classe Point qui est bien adaptée à notre programme, mais on veut maintenant que certains points aient en plus un nom. Comment faire pour :

- éviter de modifier la classe Point ?
- ne pas recopier le code de la classe Point dans une autre classe ?

```
1 public class Point {  
2     private double x, y;  
3     public Point(double x, double y) {  
4         this.x = x; this.y = y;  
5     }  
6     public void move(double x, double y) {  
7         this.x = x; this.y = y;  
8     }  
9     public String toString() {  
10        return "("+x+","+y+");"  
11    }  
12 }
```



©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

4/43

HÉRITAGE : SYNTAXE : MOT CLEF extends

★ Comment faire hériter la classe PointNommé de Point ?

Nouveaux mots-clefs :

- extends dans la signature de la classe
- super (explications plus tard slide 10)

```
1 public class PointNomme extends Point {  
2     private String name; // ne pas réécrire x et y  
3  
4     public PointNomme(double x, double y, String name) {  
5         super(x, y); // appel constructeur de Point  
6         this.name = name;  
7     }  
8  
9     public String toString() {  
10        return "PointNommé " + name + " " + super.toString();  
11    }  
12 }
```

```
21 PointNomme pnA=new PointNomme(5,6,"A");  
22 System.out.println(pnA); // "PointNommé A (5.0,6.0)"
```

Erreur courante

Attention à ne pas réécrire les attributs et les méthodes de la classe Point dans la classe PointNommé



©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

6/43

EXEMPLE : POINT ET POINTNOMMÉ

Bouger un point

```
1 Point p=new Point(1,2);
2 System.out.println(p); // (1.0,2.0)
3 p.move(3,4);
4 System.out.println(p); // (3.0,4.0)
```

Comment bouger un point nommé ?

```
5 PointNomme pnA=new PointNomme(5,6,"A");
6 System.out.println(pnA); // PointNommé A (5.0,6.0)
```

- ★ Peut-on appeler une méthode de la classe Point à partir de la variable pnA qui est de type PointNommé ?

- Oui, car grâce à l'héritage un PointNommé est un Point

```
7 pnA.move(7,8); // méthode de Point
8 System.out.println(pnA); // PointNommé A (7.0,8.0)
```

REMARQUES / VOCABULAIRE

La classe PointNommé hérite / étend / dérive de la classe Point
On dit que :

- Point est la classe mère (ou super-classe) de la classe PointNommé
- PointNommé est une classe fille (ou sous-classe) de Point

Remarques :

- Une classe fille ne peut hériter que d'une seule classe
- Une classe mère peut avoir de nombreuses classes filles
- La classe fille "connaît" sa classe mère
- La classe mère ne "connaît" pas ses classes filles

HÉRITAGE ET CONSTRUCTEURS

- ★ Que doit faire le constructeur d'une classe fille ?

Principe : 2 étapes

- 1 appeler le constructeur de la classe mère
- 2 initialiser les variables d'instance de la classe fille

```
1 public class PointNomme extends Point {
2     private String name;
3     public PointNomme(double x, double y, String name) {
4         super(x, y); // appel au constructeur de la classe mère
5         this.name = name; // init. variable d'instance fille
6     }
}
```

L'appel au constructeur de la classe mère `super(...)` :

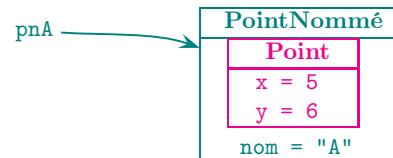
- △ doit toujours être la première instruction du constructeur
- △ ne peut être utilisé que dans un constructeur

HÉRITAGE ET PRÉSENTATION MÉMOIRE

- ★ Comment représenter un objet en mémoire quand il y a de l'héritage ?

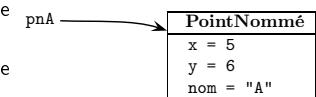
- Un objet PointNommé "englobe" un objet Point

```
PointNomme pnA=new PointNomme(5,6,"A");
```



Remarque : on peut aussi utiliser une représentation simplifiée

△ cette dernière représentation donne une vision trompeuse... donc à éviter



HÉRITAGE : SYNTAXE : MOT CLEF super

this = référence vers l'objet courant

Quand on est dans la classe fille :

`super` = permet d'accéder aux attributs, méthodes et constructeurs de la classe mère qui ne sont pas privés

- `super.maVariable` :
 - ◆ accès à la variable maVariable de la classe mère
- `super.maMéthode(...)` :
 - ◆ appel à la méthode de même signature de la classe mère
- `super(...)` :
 - ◆ appel au constructeur de la classe mère qui a la même signature

Remarque : d'autres informations sur `super` slide 40.

HÉRITAGE ET CONSTRUCTEURS

△ Il faut choisir un constructeur de la classe mère qui existe

```
1 public class Point {
2     private double x, y;
3     public Point(double x, double y) {
4         this.x = x; this.y = y;
5     }
6     public Point() {
7         this(0,0);
8     }
9 }
10 public class PointNomme extends Point {
11     private String name;
12     public PointNomme(double x, double y, String name) {
13         super(x, y); // OK Point(double, double) existe
14         this.name = name;
15     }
16     public PointNomme(String name) {
17         super(); // OK Point() existe
18         this.name = name;
19     }
20     public PointNomme(double val, String name) {
21         super(val); // Erreur le constructeur Point(double) n'existe pas dans la mère
22         this.name = name;
23     } // Solution possible : this(val, val, name);
}
```

© 2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 10/43

HÉRITAGE ET CONSTRUCTEURS

△ Bien penser à mettre le `super(...)` au début de chaque constructeur de la classe fille

Cas particulier 1 : Si on utilise `this(...)`, on n'a pas besoin de mettre `super(...)`, car cela est déjà fait par `this(...)` dans l'autre constructeur

```
10 public class PointNomme extends Point {
11     private String name;
12     public PointNomme(double x, double y, String name) {
13         super(x, y);
14         this.name = name;
15     }
16     public PointNomme(double val, String name) {
17         this(val, val, name); // pas de super(...), car déjà fait ligne 13
18     }
}
```

Cas particulier 2 : Si, dans la classe mère, il existe un constructeur accessible et sans paramètre alors il n'est pas obligatoire d'écrire `super(...)`, car cela est fait automatiquement par Java

```
31 public class Point { 36     public class PointNomme extends Point {
32     private double x,y; 37         private String name;
33     public Point() { 38             public PointNomme(String name) {
34         x=0; y=0; 39                 //super() pas obligatoire car fait automatiquement
35     } 40             this.name = name;
41 }
```

©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

HÉRITAGE ET CONSTRUCTEURS : EXEMPLE

```
1 public class Animal {
2     private String nom;
3     public Animal(String nom) {
4         this.nom = nom;
5     }
6     public String getNom() {
7         return nom;
8     }
9 }
10 public class Poule
11     extends Animal {
12     private static int cpt = 0;
13     public Poule() {
14         super("Poule"+(cpt++));
15     }
16 }
17 public class Renard
18     extends Animal {
19     private String couleur;
20     public Renard(String nom,
21                    String couleur) {
22         super(nom);
23         this.couleur=couleur;
24     }
25 }
```

△ Le nombre de paramètres des constructeurs des classes filles n'est pas forcément le même que celui de la classe mère

Le constructeur ...

- ... de la classe mère `Animal` prend 1 paramètre
- ... de la classe fille `Poule` prend 0 paramètre
- ... de la classe fille `Renard` prend 2 paramètres

```
31 Poule p1=new Poule();
32 Poule p2=new Poule();
33 Renard r=new Renard("Rox","roux");
34 System.out.println(p1.getNom());
35 // Affiche : Poule1
36 System.out.println(p2.getNom());
37 // Affiche : Poule2
38 System.out.println(r.getNom());
39 // Affiche : Rox
```

©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

14/43

RAPPEL : UML : COMPOSITION / AGRÉGATION

■ La relation de composition/agrégation est représentée par une ligne avec un losange du côté de la classe qui agrège

```
1 public class Segment{
2     private Point a,b;
3     ...
4 }
```

Diagramme de classe détaillé

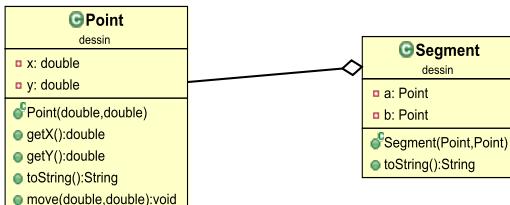
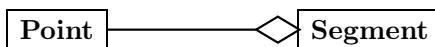


Diagramme de classe simplifié



©2021-2022 C. Marsala / V. Guigue

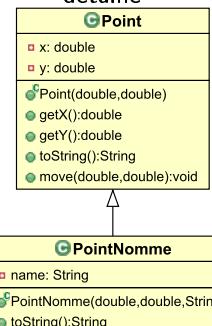
LU2IN002 - POO en Java

13/43

UML : HÉRITAGE

■ La relation d'héritage est représentée par une ligne fléchée orientée de la classe fille vers la classe mère. Le bout de la flèche est un triangle vide.

Diagramme de classe détaillé

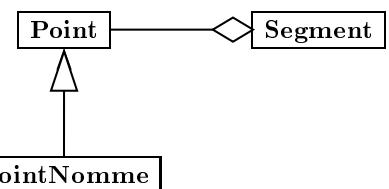


PointNomme pnA = new PointNomme(1,2,"A");

■ `pnA` est aussi un `Point`

⇒ on lisant le schéma, on voit qu'il a aussi accès à `move`, `getX`, `getY`...

Diagramme de classe simplifié



©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

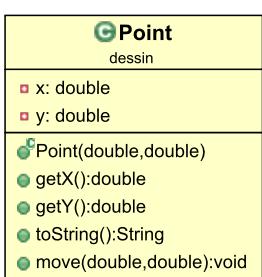
16/43

RAPPEL : UML CLIENT vs FOURNISSEUR

Pour une même classe, plusieurs types de diagrammes possibles pour plusieurs usages :

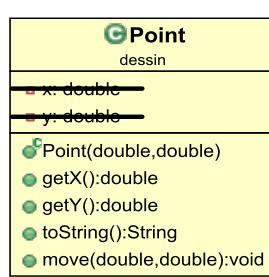
- **Vue fournisseur** : représente tous les attributs, constructeurs et méthodes
- **Vue client** : représente seulement les attributs, constructeurs et méthodes `public`

Vue fournisseur



©2021-2022 C. Marsala / V. Guigue

Vue client



LU2IN002 - POO en Java

15/43

NOUVEAU NIVEAU D'ACCÈS : protected

Attribut/méthode :

- `public` : visible partout
- `private` : visible dans la classe uniquement

Nouveau niveau d'accès : protected

- `protected` :
 - ◆ visible dans la classe
 - ◆ visible dans les classes filles et descendantes
 - ◆ mais pas dans les autres classes

Dans quel cas l'utiliser ?

- quand on veut que les classes filles puissent avoir accès à un attribut/méthode, mais pas les autres classes
- cas assez rare

Représenté par

MaClasse

```
- attPrivé : int
# attProtected : int
+ attPublic : int
- méthodePrivée()
# méthodeProtected()
+ méthodePublique()
```

©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

18/43

NOUVEAU NIVEAU D'ACCÈS : protected

```

1 public class Point {
2     private double x,y;
3     protected final int id;
4     ...
5 }
6 public class PointNomme extends Point {
7     ...
8     public void maMethode(){
9         int toto;
10        toto = x; Erreur car x private
11        toto = id; // OK car id protected
12        toto = super.id;// OK mais super pas nécessaire
13    }
14 }
15 public class Test {
16     public static void main( String [] args){
17         int toto;
18         toto = x ; Erreur car x private
19         toto = id ; Erreur car id protected
20     }
21 }

```

Une classe ⇒ 3 visions possibles : fournisseur, héritier, client

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

19/43

SUBSUMPTION

Si la classe B hérite de la classe A :

- les méthodes de A peuvent être invoquées sur un objet de la classe B

```
PointNomme pna=new PointNomme(5,6,"A");
pna.move(7,8); // méthode de Point
```

- Subsumption** : dans toute expression qui attend un A, on peut utiliser un B à la place

```
Point p = new PointNomme(1,2,"toto");
```

- ♦ p attend un Point, on peut utiliser un PointNommé à la place
- △ la variable est un Point, l'objet est un PointNommé

Polymorphisme = exploitation de la subsumption

Un PointNommé EST UN Point ⇒

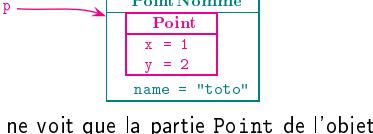
on peut le traiter comme tel

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

21/43

SUBSUMPTION : VISIONS COMPILATEURS vs JVM

```
Point p = new PointNomme(1,2,"toto");
```

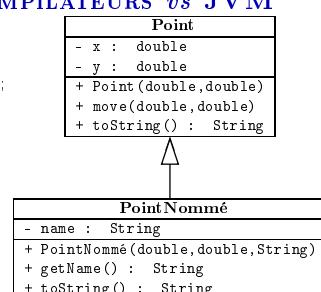


p ne voit que la partie Point de l'objet

Compilateur

La variable p est de type Point : seules les méthodes de Point sont accessibles

- p.move(1,2) // OK
- p.toString() // OK
- p.getName() ⇒ ERREUR compilation
- méthode inconnue dans Point



JVM

L'objet référencé par p est de type PointNommé

- △ p.toString() appelle la méthode de l'objet (qui est de type PointNommé)

■ Explication slide 36

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

23/43

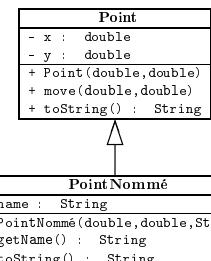
PROGRAMME

1 Héritage : syntaxe

2 Héritage : subsumption et polymorphisme

- Principe de subsumption
- Polymorphisme : application aux tableaux
- La classe Object

3 Héritage : surcharge / redéfinition



SUBSUMPTION

■ Un PointNommé EST UN Point

```
Point p = new PointNomme(1,2,"toto");
```

■ ... mais un Point N'EST PAS un PointNommé

```
PointNomme pn=new Point(2,3);
=>ERREUR compilation
```

- Par héritage, les méthodes de la classe mère sont accessibles à partir d'une variable de la classe fille

```
PointNomme pna=new PointNomme(5,6,"A");
pna.move(7,8); // OK méthode de Point
```

- ... mais les méthodes de la classe fille ne sont pas accessibles à partir d'une variable de la classe mère

```
Point p = new PointNomme(1,2,"toto"); // subsumption
p.getName(); =>ERREUR compilation car getName() pas accessible par p
```

Rôle du compilateur :

il vérifie le type des variables et les possibilités offertes par celles-ci

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

22/43

SUBSUMPTION : PARAMÈTRES DE MÉTHODE

```
1 Point p1 = new Point(1,2);
2 Point p2 = new PointNomme(1,2,"toto"); // OK
3 Point p3 = new ClasseHeritantDePoint(); // OK
```

★ Et pour les paramètres de méthode ?

```
1 public class Truc {
2     public void maMethode(Point p){
3         ...
4     }
5 } // main
6 Truc t = new Truc();
7 t.maMethode(new Point(1,2));
8 t.maMethode(new PointNomme(1,2, "toto")); // OK
9 t.maMethode(new ClasseHeritantDePoint()); // OK
```

Idée :

Comme tous les descendants de Point sont des Point...
⇒ Toutes les informations utiles/nécessaires et toutes les méthodes clientes sont disponibles

⇒ aucun problème technique en perspective !

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

24/43

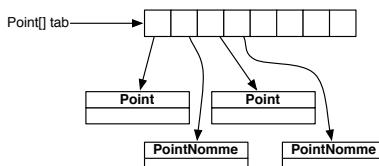
POLYMPHISME : APPLICATION AUX TABLEAUX

- ★ Application classique du polymorphisme : un tableau de points qui contient des points et des points nommés

```

1 Point [] tab = new Point [10]; // OK, 10 variables de type Point
2                                     // aucun objet Point créé
3 Point[] tab → [ ] [ ] [ ] [ ] [ ]
4
5 for(int i=0; i<tab.length; i++) {
6     if( i%2 == 0)
7         tab[i] = new Point(Math.random()*10, Math.random()*10);
8     else
9         tab[i] = new PointNomme(Math.random()*10,
                           Math.random()*10, "toto"+i);

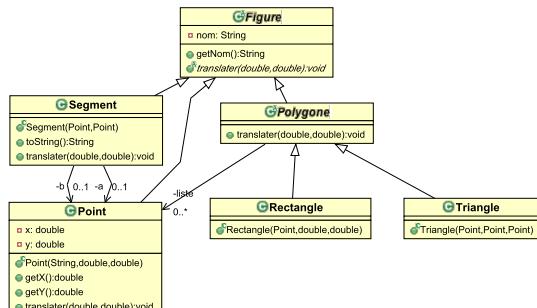
```



©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

25/43

EXEMPLE PLUS COMPLEXE



On peut :

- créer un tableau de Figure
- remplir le tableau avec des segments, points, rectangles, triangles
- translater toutes les figures du tableau
- ◆ la méthode translater est dans Figure

⇒ Bien réfléchir à la position des méthodes dans la hiérarchie

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

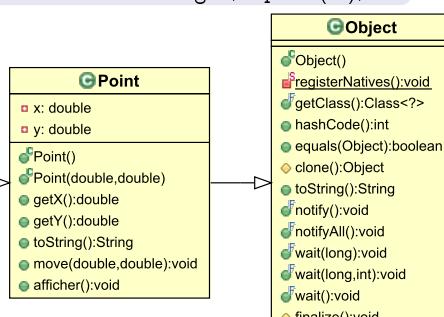
27/43

LA CLASSE Object

Classe standard

Toutes les classes dérivent de la classe **Object** de JAVA

- Héritage implicite, pas de déclaration dans la signature
`public class Point { // <=> extends Object}`
- Contient les méthodes standards `toString()`, `equals(...)`, ...

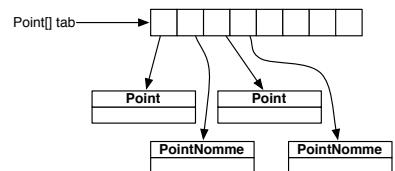


©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

29/43

POLYMPHISME : APPLICATION AUX TABLEAUX

- ★ Peut-on bouger tous les points du tableau ?



Oui, car la méthode move est dans Point

```

11 for(int i=0;i<tab.length; i++) {
12     tab[i].move(1,2);
13 }

```

```

14 // de même, avec la boucle sans indice
15 for(Point p : tab) {
16     p.move(1,2);
17 }

```

⇒ Les instances de Point et les instances de PointNomme ont bougé

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

26/43

LIMITES DU POLYMPHISME

- △ On ne peut invoquer que les méthodes de la classe mère
- Exemple : si le type est Figure, on ne peut invoquer que les méthodes de Figure, même si l'objet est un Point

```

1 Figure [] tabFig = new Figure[10];
2 tabFig[0]=new Point();
3 tabFig[0].translater(2,2); // OK variable de type Figure
4                                         // et translater(x,y) dans Figure
5 tabFig[0].getX(); ERREUR variable de type Figure
6                                         => getX() méthode de Point pas accessible
7                                         même si l'objet est un Point

```

- Pour pouvoir accéder aux méthodes de la classe fille, il faut caster :

```

8 ((Point)tabFig[0]).getX(); // OK
9 // Autre solution :
10 Point p=(Point)tabFig[0]; p.getX(); // OK

```

On l'étudiera plus tard...

Rôle du compilateur :

il vérifie le type des variables et les possibilités offertes par celles-ci

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

28/43

LA CLASSE Object (SUITE)

- Une variable de type **Object** peut référencer un objet de type quelconque

```

1 Object o = new Point(1,2);
2 Object o2 = new PointNomme(2,3,"toto");

```

- ... mais on ne peut (presque) rien faire sur o et o2

```

3 o.toString() // OK
4 o2.toString() // OK
5 o.move(1,2) // KO
6 o2.getName() // KO

```

- Crédation d'un tableau contenant "n'importe quoi"

```

7 Object [] tab = new Object[10];
8 tab[0] = "toto";
9 tab[1] = 10; // --> conversion implicite en Integer
10 tab[2] = new Point(1,2);
11 tab[3] = new PointNomme(2,3,"toto");

```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

30/43

PROGRAMME

- 1 Héritage : syntaxe
- 2 Héritage : subsumption et polymorphisme
- 3 Héritage : surcharge / redéfinition
 - Surcharge et héritage
 - Redéfinition
 - Redéfinition et usage de super
 - Ouverture de la redéfinition

SURCHARGE : QUI FAIT QUOI

Le compilateur (pré)-sélectionne les méthodes :

- Ces méthodes sont totalement différentes pour le compilateur qui analyse le type des paramètres

```
1 public class Point {  
2     ...  
3     public void move(double dx, double dy){ // 1  
4         x+=dx; y+=dy;  
5     }  
6     public void move(double dx, double dy, double scale){ // 2  
7         x+=dx*scale; y+=dy*scale;  
8     }  
  
9     Point p = new Point(1,2);  
10    p.move(3, 1); // présélection de 1  
11    p.move(3, 1, 0.5); // présélection de 2
```

SURCHARGE ET HÉRITAGE

- Les méthodes peuvent être indifféremment dans la classe fille ou dans la classe mère

```
1 public class Point {  
2     ...  
3     public void move(double dx, double dy) { // 1  
4         x+=dx; y+=dy;  
5     }  
6 }  
7 public class PointNomme extends Point {  
8     ...  
9     public void move(double a) { // 2  
10        super.move(a,a);  
11    }  
12 }  
  
13 PointNomme pn=new PointNomme(1,2,"toto");  
14 pn.move(3,4); // 1 : méthode héritée de Point  
15 pn.move(5); // 2 : méthode de PointNomme
```

★ Que se passe-t-il quand une méthode de la classe fille a la même signature qu'une méthode de la classe mère ?

⇒ redéfinition de méthode ≠ surcharge de méthode

RAPPEL : SURCHARGE DE MÉTHODES

Définition

- Même nom de méthode MAIS paramètre(s) différent(s)
- Le type de retour n'est pas considéré pour différencier les méthodes

```
1 public class Point {  
2     ...  
3     public void move(double dx, double dy){  
4         x+=dx; y+=dy;  
5     }  
6     public void move(double dx, double dy, double scale){  
7         x+=dx*scale; y+=dy*scale;  
8     }  
9     public void move(int dx, int dy){  
10        x+=dx; y+=dy;  
11    }  
12    public void move(Point p){  
13        x+=p.x; y+=p.y;  
14    }
```

SURCHARGE : LES LIMITES

```
1     public void move(double dx, double dy){  
2         x+=dx; y+=dy;  
3     }
```

- Interdiction d'avoir des signatures identiques

```
4     // Même signature pour le compilateur  
5     // => ERREUR de compilation  
6     public void move(double a, double b){  
7         x+=a; y+=b;  
8     }
```

- Pas de prise en compte du type de retour

```
9     // Même signature pour le compilateur  
10    // => ERREUR de compilation  
11    public Point move(double dx, double dy){  
12        x+=dx; y+=dy;  
13        return this;  
14    }
```

REDÉFINITION

Définition

Redéfinition d'une méthode de **même signature** dans la classe fille

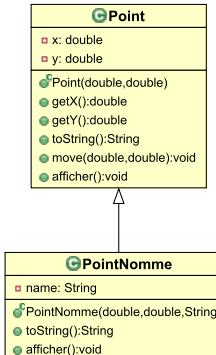
```
1 public class Point {  
2     ...  
3     public void afficher() { // 1  
4         System.out.println("Je suis un Point");  
5     }  
6 }  
7 public class PointNomme extends Point {  
8     ...  
9     public void afficher() { // 2  
10        System.out.println("Je suis un PointNomme");  
11    }  
12 }
```

- Pas de problème à la compilation
- A l'exécution, la JVM décide de la méthode à invoquer en fonction du type de l'instance appelante
- Exemple classique : la méthode `toString()` de `Object` est souvent redéfinie dans les classes filles

REDÉFINITION : EXEMPLES DE FONCTIONNEMENT

Cas 1 : facile

```
1 Point p = new Point(1, 2);
2 p.afficher();
```



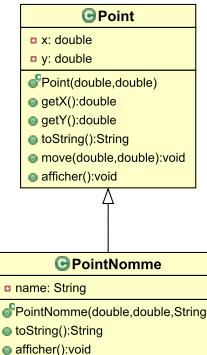
Dans la classe Point, une seule méthode correspond à la signature afficher()
Affichage de :

Je suis un Point

REDÉFINITION : EXEMPLES DE FONCTIONNEMENT

Cas 2 : résolution d'une ambiguïté

```
1 PointNomme pn = new PointNomme(1, 2, "toto");
2 pn.afficher();
```



- Dans la classe PointNommé, deux méthodes correspondent à la signature afficher() :
 - une dans Point
 - une dans PointNommé

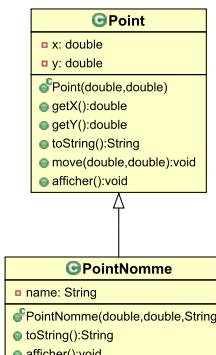
- La JVM choisit, au moment de l'exécution du programme en fonction du type de l'objet référencé par pn

- Affichage de :
Je suis un PointNomme car l'objet est un PointNommé

REDÉFINITION : EXEMPLES DE FONCTIONNEMENT

Cas 3 : Redéfinition + subsumption

```
1 Point p = new PointNomme(1, 2, "toto"); // subsumption
2 p.afficher(); // ???
```



- Compilation** : vérification du type de la variable uniquement
 - variable p est un Point
 - afficher() existe dans Point
⇒ compilation OK
- Exécution** : La JVM choisit, au moment de l'exécution en fonction type de l'objet référencé par p
 - p référence un objet de type PointNomme
 - recherche de afficher() dans PointNomme
- Affichage de :**
Je suis un PointNomme car l'objet est un PointNommé

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 37/43

REDÉFINITION ET USAGE DE super 1/3

Le mot clef **super** permet :

REDÉFINITION ET USAGE DE super 1/3

Le mot clef **super** permet :

- de préciser que l'on utilise des informations de la super-classe

```
1 public class PointNomme extends Point {
2 ...
3     public void afficher(){
4         System.out.println("Je suis un PointNomme" +
5             " de coordonnées : " + super.getX() + " " +
6             super.getY());
7     }
8 }
```

- de forcer le programme à aller chercher une méthode dans la super-classe (**obligatoire**)

```
1 public class PointNomme extends Point {
2 ...
3     public void affichageGlobal(){
4         afficher(); // → Je suis un PointNomme
5         this.afficher(); // → Je suis un PointNomme
6         super.afficher(); // → Je suis un Point
7     }
8 }
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 40/43

REDÉFINITION ET USAGE DE super 2/3

L'un des usages les plus classiques concerne **toString()** :

```
1 // classe Point
2     public String toString() {
3         return "("+x+","+y+")";
4     }
5 // classe PointNomme
6     public String toString() {
7         return "PointNomme" + name + " " + super.toString();
8     }
```

1 Quels sont les affichages en sortie du code suivant :

```
21 Point p = new Point(1,2);
22 PointNomme pnA = new PointNomme(3,4, "A");
23 Point pNb = new PointNomme(5,6, "B");
24 System.out.println(p.toString());
25 // (1,2)
26 System.out.println(pnA.toString());
27 // PointNomme A (3,4)
28 System.out.println(pNb.toString());
29 // PointNomme B (5,6) // toString() de l'objet, pas de la variable
```

2 Que se passerait-il si on oublie le super à la ligne 7 ?

- Un appel récursif...

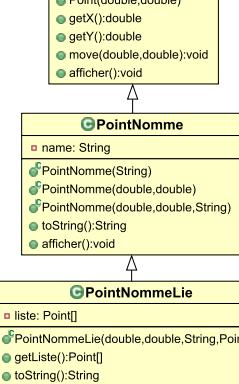
©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 41/43

REDÉFINITION ET USAGE DE super 3/3

Ajout d'une classe "petite-fille" PointNommeLie pour un point lié à d'autres dans l'espace (graphe)

Dans la classe PointNommeLie :

- toString() ; // OK PointNommeLie
- super.toString() ; // OK PointNomme
- super.super.toString() ; // syntaxe interdite



- △ Avec super, on ne peut remonter que de un niveau
- getX() ; // OK, existe ici par héritage de Point
- super.getX() ; // OK existe dans PointNomme par héritage de Point

Les méthodes redéfinies dans la classe mère bloquent l'accès aux versions de la classe "grand-mère"

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 42/43

OUVERTURE DE LA REDÉFINITION

Définition

Il est possible d'**augmenter** la visibilité d'une méthode dans la classe fille mais **pas de la réduire**

Exemple :

- dans Point

```
1 protected Point maMethode(){ return new Point(...); }
```

- dans PointNommé

```
2 // pour éviter le cast  
3 protected PointNommé maMethode(){ return new PointNomme(...); }
```

- dans PointNommeLie

Ouverture de la redéfinition possible (méthode devient **public**)

```
4 public PointNommeLie maMethode(){ return new PointNommeLie(...); }
```

Mais pas de réduction de la visibilité possible

```
5 private PointNommeLie maMethode(){ return new PointNommeLie(...); }
```

```
6 // ERREUR : redéfinition : ne peut pas devenir private
```

LU2IN002 - Introduction à la programmation orientée objet

Responsable de l'UE et du cours du vendredi :
Christophe Marsala
(e-mail : Christophe.Marsala@lip6.fr)

Cours du lundi : Sabrina Tollari
(e-mail : Sabrina.Tollari@lip6.fr)

(support réalisé à partir de ceux de Christophe Marsala et de Vincent Guigue)



Cours 7 – lundi 16 octobre 2023

PROGRAMME

1 Héritage : classes et méthodes abstraites

- Mot clef abstract

2 Héritage : divers

3 Héritage : classes et méthodes final

PROGRAMME DU JOUR

1 Héritage : classes et méthodes abstraites

- Mot clef abstract

2 Héritage : divers

- Opérateur instanceof
- Méthode getClass()
- Héritage et cast
- Méthode equals : égalité entre objets
- Méthode clone() : copie d'objets

3 Héritage : classes et méthodes final

NOUVEAUX CONCEPTS

■ Classe abstraite

- Classe qui ne sera pas instanciable
- Les classes filles pourront être instanciables
- Exemple :
 - Animal (abstraite) : définit un comportement général
 - Mouton, Tigre : animaux avec comportements spécifiques

■ Méthode abstraite

- Seulement dans les classes abstraites
- Contient une signature mais pas de corps de méthode (le code de la méthode). Exemple :
 - Animal (abstraite) contient la signature :
String régimeAlimentaire();
- La définition du corps de la méthode sera fait dans les classes filles (ou descendantes). Exemples :
 - Mouton : { return "herbivore"; }
 - Tigre : { return "carnivore"; }



©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

4/38

CLASSE ET MÉTHODE ABSTRAITE

★ Quand définir une méthode abstraite ?

Principe

- Un concept unificateur qui permet de factoriser du code pour toutes les classes qui hériteront. Exemple :
 - Tous les animaux (moutons, tigres...) ont un nom
 - On peut factoriser nom, getNom(), ... dans la classe Animal
 - Tous les animaux ont un régime alimentaire
 - Problème : ce régime dépend des classes filles
 - La signature de la méthode régimeAlimentaire() peut être factorisée dans la classe Animal, mais pas le corps de la méthode qui dépend des filles ⇒ méthode abstraite
- Introduction de la notion de contrat : toutes les classes filles devront gérer ce qui est décidé par la classe mère (signature de méthode abstraite). Exemple :
 - Chaque classe fille de Animal doit définir le corps de la méthode régimeAlimentaire() ou bien être abstraite

CLASSE ET MÉTHODE ABSTRAITE : EXEMPLE

```
1 public abstract class Animal { //=> new Animal("A1"); Erreur abstract
2     private String nom;
3     public Animal(String nom) {
4         this.nom=nom;
5     }
6     public String getNom() { return nom; }
7     public abstract String régimeAlimentaire(); △ signature
8 }                                            => pas d'accolades
9 public class Mouton extends Animal {          // main
10    public Mouton(String nom) {                25 // tab=[]
11        super(nom);                         26 new Mouton("M1"),
12    }                                         27 new Mouton("M2"),
13    public String régimeAlimentaire(){        28 new Tigre("T1")
14        return "végétarien"; // code          29 };
15    }                                         30 }
16 }                                            31
17 public class Tigre extends Animal {           32 for(Animal a : tab) {
18    public Tigre(String nom) {               33 System.out.println(
19        super(nom);                      34     a.getNom()+" "+
20    }                                         35     a.régimeAlimentaire());
21    public String régimeAlimentaire(){      36 }
22        return "carnivore"; // code          37 }
23    }                                         38 }
24 }
```

Affichage :
M1 végétarien
M2 végétarien
T1 carnivore

CLASSE ET MÉTHODE ABSTRAITE : SYNTAXE

```
1 public abstract class Figure {
2     public Figure() { }
3     public abstract String getTypeFigure(); // signature seulement
4 } // pas d' accolades
```

On ne peut pas créer d'instance de la classe Figure

```
new Figure(); // ERREUR compilation : la classe est abstract
```

Les classes qui héritent de Figure devront (2 cas possibles) :

A soit implémenter getTypeFigure()

◆ Si on connaît le type de la figure dans la classe

```
public class Point extends Figure {
    ...
    public String getTypeFigure() { // code de la méthode
        return "Point"; // dépend de chaque classe fille
    }
}
```

B soit être elles-mêmes abstraites

◆ Si on ne connaît pas le type de la figure dans la classe. Pour Polygone, on ne sait pas si c'est un rectangle, un triangle, ...

```
public abstract class Polygone extends Figure {
    ...
}
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 7/38

PROPRIÉTÉS DES CLASSES ABSTRAITES

Les classes abstraites sont des classes comme les autres.

Elles peuvent avoir :

- des attributs
- des constructeurs
- des méthodes

mais en plus elles peuvent avoir des méthodes abstraites.

```
1 public abstract class Figure { // abstract
2     private double x, y;
3     public Figure(double x, double y) {
4         this.x=x; this.y=y;
5     }
6     public void move(double x, double y) {
7         this.x=x; this.y=y;
8     }
9     public abstract String getTypeFigure(); // abstract
10 }
```

Idée

Les classes abstraites sont pensées pour leurs descendantes, les classes filles qui en seront dérivées

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 9/38

CLASSE ABSTRAITE VERSUS CONSTRUCTEUR PRIVÉ

```
1 public abstract class Animal {
2     public Animal() { }
3     public static void maMethode() {
4         new Animal(); Erreur car la classe
5     } // est abstract
6 }
7 public class Mouton extends Animal {
8     public Mouton() {
9         super(); //OK
10 }
11 }
12 // main de la classe Test
13 new Animal(); Erreur car la
14 classe Animal est abstract
15 new Mouton(); //OK

Une classe abstraite n'est jamais instanciable, mais sa classe fille peut être instanciée

16 public class Alea {
17     private Alea() { }
18     public static void maMethode() {
19         new Alea(); //OK instantiation
20     }
21 }
22 public class FilleAlea extends Alea {
23     public FilleAlea() {
24         super(); Erreur car le constructeur
25     } de la mère est private
26 }

// main de la classe Test
new Alea(); Erreur car le
constructeur de Alea est privé

Une classe avec un (seul) constructeur privé ne peut pas être instanciée dans une autre classe
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 11/38

CLASSE ET MÉTHODE ABSTRAITE : SYNTAXE

Dans le cas B :

```
1 public abstract class Figure {
2     public Figure() { }
3     public abstract String
4             getTypeFigure();
5 }
6 public abstract class Polygone
7     extends Figure {
8     ...
9 }
```

Ce sont les classes descendantes (Rectangle, Triangle, ...) qui devront définir le code de la méthode

```
10 public class Rectangle extends Polygone {
11     ...
12     public String getTypeFigure() {
13         return "Rectangle"; // code
14     }
15 }
```

Remarque : on peut avoir plusieurs méthodes abstraites déclarées et définies à différents niveaux de la hiérarchie (exemple : translator)

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 15/38

RÉSUMÉ : CLASSE ET MÉTHODE ABSTRAITE

Résumé de quelques règles à connaître :

- 1 une classe abstraite ne peut être instanciée
- 2 si une méthode est abstraite, alors sa classe doit être abstraite
- 3 si une classe hérite d'une méthode abstraite, elle doit :
 - ▲ soit définir le code de la méthode
 - soit être déclarée abstraite. Ce seront les classes descendantes qui définiront le code de la méthode

△ Une classe abstraite peut ne pas contenir de méthode abstraite

★ Pourquoi déclarer une classe abstraite si elle ne contient pas de méthode abstrait ?

- si on veut empêcher la création d'instance
- si la classe représente une notion abstraite (= concrète) pour notre problème
 - ◆ Exemple : Animal est une notion abstraite par rapport à Tigre ou Mouton qui sont des notions concrètes pour notre problème
 - ⇒ pas de sens de créer un objet Animal seulement

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 10/38

(RETOUR) SUR LES BONNES PRATIQUES

Développement à long terme

Modification d'un projet existant = ajout d'une classe

- ne pas modifier les classes existantes
- ajouter des classes filles

Idée

Structurer un projet avec des classes abstraites =

- les classes filles possèdent des fonctionnalités dès leur création
- ajout de **contraintes** sur les classes filles
 - ◆ **plus facile** à développer (classe fille = canevas à remplir)
 - ◆ **contrat** sur les fonctionnalités (garanties)
 - ◆ garanties sur des **classes qui n'existent pas encore** : facilités d'évolution du code
- usage du polymorphisme
 - ◆ ex. : tableau hétérogène ⇒ + de possibilités

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

12/38

PROGRAMME

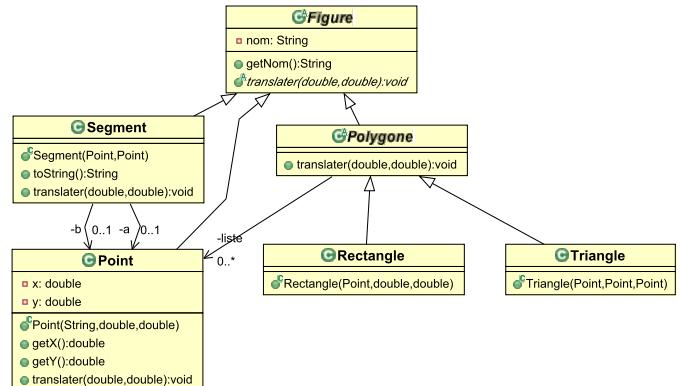
1 Héritage : classes et méthodes abstraites

2 Héritage : divers

- Opérateur instanceof
- Méthode getClass()
- Héritage et cast
- Méthode equals : égalité entre objets
- Méthode clone() : copie d'objets

3 Héritage : classes et méthodes final

EXEMPLE DU LOGICIEL DE DESSIN



RÉCUPÉRER LE TYPE D'UN OBJET DYNAMIQUEMENT

Cas amusant :

le type des objets est parfois (souvent) inconnu du développeur

Exemple :

```

1 Figure fig;
2 if (Math.random() > 0.5)
3     fig = new Point();
4 else
5     fig = new Segment();
  
```

- Pour le compilateur, la syntaxe est correcte :
 - dans tous les cas, **fig** référence un objet qui est une **Figure**
- Pour la JVM, quel est le type de l'objet référencé par **fig** ?
 - Cela dépend de l'exécution...

RÉCUPÉRER LE TYPE D'UN OBJET DYNAMIQUEMENT

- Le compilateur vérifie (**statiquement**) le **type des variables**

- Comment connaître le **type d'un objet** à l'exécution (**dynamiquement**) ? 2 moyens :

- Opérateur **instanceof**
- Méthode **getClass()**

- Comment revenir au type initial de l'objet pour accéder aux méthodes spécifiques ?

- Utiliser un **cast**

```

1 Figure fig = new Point();
2 fig.methodeDePoint(); // Erreur compilation
3 ((Point)fig).methodeDePoint(); // OK
4 // OU en 2 instructions :
5 Point p = (Point) fig;
6 p.methodeDePoint(); // OK
  
```

OPÉRATEUR instanceof

var instanceof NomClasse

⇒ retourne un boolean

- Retourne true si l'objet référencé par la variable var est une instance de la classe **NomClasse**
- Retourne false sinon (en particulier si var est null)

```

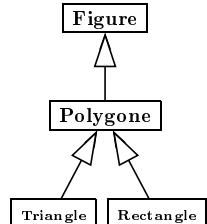
1 Figure fig;
2 if (Math.random() > 0.5)
3     fig = new Point();
4 else
5     fig = new Segment();
6
7 if (fig instanceof Point)
8     System.out.println("C'est un Point (ou descendante de Point)");
9 else
10    System.out.println("Ce n'est pas un Point");
  
```

OPÉRATEUR instanceof

Figure f1 = new Rectangle(); // suboption

Quelle est la valeur des expressions suivantes ?

- f1 instanceof Rectangle // true
- f1 instanceof Triangle // false
- f1 instanceof Polygone // true
- f1 instanceof Figure // true



⇒ L'objet est un rectangle ... qui est un polygone... qui est une figure, mais n'est pas un triangle

Il faut comprendre instanceof comme «EST UN?»

Figure f2 = null;
f2 instanceof Figure // false : pas d'objet

Polygone plg = (Polygone) f1;
plg instanceof Rectangle // true : l'objet est un rectangle
plg instanceof Figure // true : l'objet est un rectangle qui est une figure

OPÉRATEUR instanceof : DISCUSSION (1/2)

⚠️ Attention à ne pas mal utiliser instanceof

Exemple : il ne faut pas utiliser instanceof dans une méthode...

- ... de la classe mère pour connaître ses classes filles
- ... quelconque qui distingue toutes les filles connues

★ Pourquoi ?

```
1 public static void afficheType(Figure f) {  
2     if(f instanceof Point)  
3         System.out.println("C'est un Point");  
4     else if(f instanceof Segment)  
5         System.out.println("C'est un Segment");  
6     else if(f instanceof Rectangle)  
7         System.out.println("C'est un Rectangle");  
8     // etc ... un if par classe fille de Figure connue  
9 }
```

Que se passe-t-il si on écrit une nouvelle classe fille de Figure ?

⇒ La méthode devient fausse.

MÉTHODE getClass()

Méthode getClass()

- Méthode de la classe Object
- S'utilise sur une instance (syntaxe différente de instanceof)
- Retourne la classe de l'instance

```
1 Figure fig = new Segment();  
2 System.out.println("fig est de type " + fig.getClass());  
3 // Affiche : fig est de type class Segment
```

Usage classique pour comparer le type de deux instances :

```
1 // soit deux objets obj1 et obj2  
2 if (obj1.getClass() != obj2.getClass())  
3 ...
```

HÉRITAGE ET CAST

Cast = 2 modes de fonctionnement

- Conversion sur les types basiques : le codage des données change. Souvent implicite dans votre codage...

```
1 double d = 1.4;  
2 int i = (int) d; // i=1
```

- Conversion dans les hiérarchies de classes :

```
3 Figure fig = new Point(); // subsumption  
4 Point p1 = fig; // Erreur compilation  
5 Point p2 = (Point) fig; // OK
```

⚠️ le cast ne modifie pas l'objet

Le cast convertit la référence à un objet de type classe A en une référence à un objet de type classe descendante de A.

Caster c'est un peu comme ça le programmeur disait au compilateur : "je sais que normalement je n'ai pas le droit de faire cette affectation, mais fait moi confiance" ... cependant le programmeur peut se tromper...

OPÉRATEUR instanceof : DISCUSSION (2/2)

Bonne façon de résoudre ce genre de problème

Utiliser une méthode abstract dans la classe mère pour imposer aux filles d'écrire le code de la méthode

- dans Figure :

```
1 public abstract String getTypeFigure();
```

- dans Point :

```
2 public String getTypeFigure() { return "Point"; }
```

- dans Rectangle :

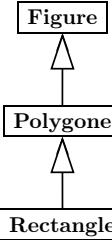
```
3 public String getTypeFigure() { return "Rectangle"; }
```

- Code générique (éventuellement en dehors des classes) :

```
4 public static void afficheType(Figure f) {  
5     System.out.println("C'est un " + f.getTypeFigure());  
6 }
```

COMPARAISON getClass() ET instanceof

⚠️ Ne pas confondre getClass() et instanceof



- instanceof est un opérateur qui indique si l'objet est de type MaClasse ou ses descendantes

```
Figure f1 = new Rectangle(); // subsumption  
♦ f1 instanceof Rectangle // true  
♦ f1 instanceof Polygone // true  
♦ f1 instanceof Figure // true
```

- getClass() est une méthode qui retourne la classe exacte de l'objet

```
Figure figR1 = new Rectangle();  
Figure figR2 = new Rectangle();  
Figure figP = new Point();
```

```
figR1.getClass() == figR2.getClass() // true  
figR1.getClass() == figP.getClass() // false
```

CAST : LIMITES

- Cast inutile : subsumption

```
1 Point p1 = new Point();  
2 Figure fig1 = (Figure)p1; // OK mais cast inutile  
3 Figure fig2 = p1; // OK subsumption
```

- Cast obligatoire : comment revenir à un Point ?

```
4 Point p2 = fig2; // Erreur compilation  
5 Point p3 = (Point)fig2; //OK compilation cast obligatoire  
6 //OK exécution l'objet est un point
```

- Mauvais cast

```
7 Figure fig3 = new Segment();  
8 Point p4 = (Point)fig3; // compilation OK (!)
```

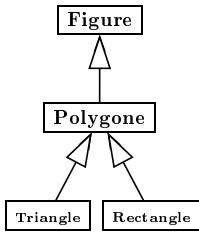
Exécution : Crash du programme avec le message suivant

Exception in thread "main" java.lang.ClassCastException:
Segment cannot be cast to Point

A l'exécution, l'objet qui est un Segment, ne peut pas être affecter à une variable de type Point, car un segment n'est pas un point (comparer avec le cas à la ligne 5 où l'objet est un point).

CAST : AVEC PLUSIEURS NIVEAUX DE HIÉRARCHIE

Exemples de cas possibles quand on a 3 niveaux d'héritage



```
1 Figure fig1 = new Rectangle();
2 Polygone poly1 = new Rectangle();
3 
4 Rectangle rect1 = (Rectangle) fig1; // OK
5 Rectangle rect2 = (Rectangle) poly1; // OK
6 
7 Polygone poly2 = (Polygone) fig1; // OK
8 
9 Figure fig2 = poly2; // OK subsumption
10 
11 Rectangle rect3=(Rectangle) fig2; // OK
```

Exemples de cas impossibles :

```
1 Figure f1 = new Rectangle();
2 Triangle t1 = (Triangle) f1; // OK compilation
3 // ERREUR exécution ClassCastException : Rectangle cannot be cast to Triangle
4 
5 Rectangle r1=new Rectangle();
6 Triangle t2 = (Triangle) r1; // ERREUR compilation !!!
7 // incompatible types: Rectangle cannot be converted to Triangle
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

25/38

MÉTHODE equals : ÉGALITÉ ENTRE OBJETS

Comparer deux objets

Pour comparer deux objets, il faut utiliser la méthode `equals` de la classe `Object` qui a pour signature :

```
boolean equals(Object obj)
```

- Par défaut, `equals` teste l'égalité référentielle, ce qui n'est pas intéressant...

```
1 Point p1=new Point(1,2);
2 Point p2=new Point(1,2);
3 Point p3=p1;
4 p1.equals(p2); // false : pas le même objet
5 p1.equals(p3); // true : même référence (même objet)
```

- Solution : **redéfinir** la méthode dans la classe fille pour tester l'égalité des variables d'instances

- △ Il faut que la signature soit **exactement la même**, en particulier **il faut que le paramètre soit de type Object**
- △ Si le paramètre n'est pas de type `Object`, ce n'est pas de la redéfinition, mais de la surcharge (à éviter pour `equals`)!!!

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

27/38

MÉTHODE equals

Méthode equals

Il y a toujours un **cast** (**sécurisé**) dans `equals` pour pouvoir accéder aux attributs à comparer

Exemple sur la classe `Point` :

```
1 public boolean equals(Object obj) { // V1
2     if (this == obj)
3         return true;
4     if (obj == null)
5         return false;
6     if (getClass() != obj.getClass())
7         return false;
8     Point other = (Point) obj;
9     if (x != other.x)
10        return false;
11     if (y != other.y)
12        return false;
13     return true;
14 }
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

29/38

CAST : SÉCURISATION

Idée

Vérifier le type de l'instance avant la conversion

En général, pour éviter les erreurs de cast à l'exécution, il est préférable de tester le type de l'instance avant de caster.

```
1 Figure f = new Segment();
2 f.methodeDeSegment(); // ERREUR compilation
3 Segment s;
4 if(f instanceof Segment) { // bon usage de instanceof
5     s = (Segment) f;
6     s.methodeDeSegment(); // OK
7 }
```

⇒ vous utiliserez **systématiquement** cette sécurisation

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

26/38

MÉTHODE equals

Exemple pour la classe `Point`.

Cette méthode peut en particulier réaliser les tests suivants :

- 1 Vérifier s'il y a égalité référentielle

```
1 public boolean equals(Object obj) {
2     if (this == obj) return true; // optionnel
```

- 2 Vérifier le type de l'object référencé par le paramètre `obj`

```
3     if (obj == null) return false;
4     if (getClass() != obj.getClass()) return false;
```

- 3 Caster la référence de l'object `obj` pour atteindre ses attributs

```
5     Point other = (Point) obj;
```

- 4 Vérifier l'égalité entre attributs

```
6         if (x != other.x) return false;
7         if (y != other.y) return false;
8     }
9 }
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

28/38

MÉTHODE equals : getClass vs instanceof

Imaginons la redéfinition suivante de `equals` utilisant `instanceof` au lieu de `getClass()` :

```
1 public boolean equals(Object obj) { // V2
2     if (this == obj)
3         return true;
4     if (!(obj instanceof Point))
5         return false;
6     Point other = (Point) obj;
7     if (x != other.x)
8         return false;
9     if (y != other.y)
10        return false;
11    }
12 }
```

Quelles sont les limites de l'implémentation v2?

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

30/38

MÉTHODE equals : getClass vs instanceof

```
1 Point p = new Point(1,2);
2 PointNommé pn = new PointNommé("toto",1,2);
3
4 if(p.equals(pn))
5     System.out.println("ils sont égaux !!!");
6 else
7     System.out.println("ils ne sont pas égaux !!!");
```

- V1 : pas égaux
- V2 : égaux (car PointNommé est une instance de Point).
Est-ce légitime qu'ils soient égaux ?

Autre problème : si on redéfinit equals dans PointNommé (il faut qu'elle appelle la méthode equals de sa classe mère) alors :

- p.equals(pn) appelle la méthode equals de Point
 - ◆ V2 : égaux (pn instance de Point)
 - pn.equals(p) appelle la méthode equals de PointNommé
 - ◆ V2 : pas égaux (p n'est pas instance de PointNommé)
- ⚠ Problème de symétrie : p.equals(pn) ≠ pn.equals(p)

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

31/38

MÉTHODE equals

- La méthode equals est dans Object ⇒ Tous les objets sont comparables entre eux !

- On peut donc rechercher un objet en particulier dans un tableau (ou une ArrayList) exploitant le polymorphisme :

```
1 ArrayList<Object> al = new ArrayList<Object>();
2 al.add("toto");
3 al.add(10);
4 al.add(new Point(1,2));
5 al.add(new PointNommé("A",3,4));
```

- Sachant que la méthode contains de la classe ArrayList utilise equals. Quelle est la valeur de :

```
al.contains(new Point(1,2))
```

- ◆ si la méthode equals n'a pas été redéfinie dans Point ?
 - false car la méthode equals de Object test l'égalité référentielle
- ◆ si la méthode equals a été redéfinie dans Point ?
 - true car la méthode equals de Point test l'égalité structurelle

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

32/38

COPIE D'OBJETS : MÉTHODE clone()

Solution 1 : constructeur de copie (cours 3)

Solution 2 : méthode standard clone()

Méthode standard de la classe Object dont l'objectif est de retourner un nouvel objet qui est une copie du l'objet courant

- Exemple de code dans la classe Point

```
1 public class Point{
2 ...
3     public Point clone(){
4         return new Point(x, y);
5     }
6 }
```

- Usage :

```
1 Point p1 = new Point(1,2);
2 Point p2 = p1.clone();
```

- Comparaison constructeur de copie et méthode clone()

- ◆ Résultat ABSOLUMENT identique
- ◆ Cas d'utilisation un peu différent

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

33/38

PROGRAMME

1 Héritage : classes et méthodes abstraites

2 Héritage : divers

3 Héritage : classes et méthodes final

MÉTHODE clone() ET REDÉFINITION

Rappel : il est possible d'**augmenter la visibilité** d'une méthode dans la classe fille mais **pas de la réduire**

- dans Object : la méthode est protected

```
protected Object clone(){...}
```

- dans Point

```
// pour éviter les cast : le type de retour est Point
public Point clone(){return new Point(...);}
```

- dans PointNommé

```
public PointNommé clone(){return new PointNommé(...);}
```

```
1 Point [] tab={new Point(1,2),new PointNommé("A",3,4)};
2 Point [] tabCopie=new Point[tab.length];
3 for(int i=0;i<tab.length;i++)
4     tabCopie[i]=tab[i].clone();
```

Remarque : quand il y a de l'héritage, on ne peut pas faire cela avec des constructeurs de copie

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

34/38

RAPPEL : final POUR LES ATTRIBUTS

Idée : protéger ses objets... Et ses programmes

Initialiser les valeurs des attributs sans pouvoir les modifier ensuite

Exemple : String

```
1 public class Point{
2     public final double x,y;
3     public Point(double x, double y){
4         this.x = x; this.y = y;
5     }
6 ...
7 }
```

- Interdiction de modifier x et y dans les méthodes

- ◆ pas de setter, pas de translation...

- Modification d'un Point = création d'une nouvelle instance

- Possibilité de laisser les attributs public... Puisque non modifiable

- Sécurité lorsqu'un objet est passé en argument de méthode

SCiences
Sorbonne
Université

©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

36/38

RAPPEL : LES CONSTANTES

Idée

Pour sécuriser le code, interdisons les modifications de certaines valeurs (notamment les constantes).

- final = sécurisation = impossibilité de modifier
- static = indépendante des instances
- Usage : une constante est définie en majuscule

```
1 public class MaClasse{  
2     public final static int MA_CONSTANTE = 10;  
3 }
```

Usage :

- constantes universelles (Color.RED, Color.YELLOW, Math.PI, Double.POSITIVE_INFINITY...)
- typologie (type de codage d'un pixel, organisation du BorderLayout)...
- bornes algorithmiques (NB_ITER_MAX, TAILLE_MAX...)

final : USAGES LIÉS À L'HÉRITAGE

- Méthode final : ne peut pas être redéfinie dans les classes filles

```
1 public class Point {  
2     ...  
3     public final double getX(){ ... }  
4 }  
5  
6 public class PointNomme extends Point{  
7     ...  
8     // Compilation impossible : méthode de la classe mère final  
9     public double getX(){ ... }  
10 }
```

- Classe final : ne peut pas être héritée
(exemples : String, Integer, Double...)

```
1 public final class Point {  
2     ...  
3 }  
4  
5 // Compilation impossible : classe "mère" final  
6 public class PointNomme extends Point {  
7     ...  
8 }
```

LU2IN002 - Introduction à la programmation orientée objet

Responsable de l'UE et du cours du vendredi :
Christophe Marsala
(e-mail : Christophe.Marsala@lip6.fr)

Cours du lundi : Sabrina Tollari
(e-mail : Sabrina.Tollari@lip6.fr)

(support réalisé à partir de ceux de Christophe Marsala et de Vincent Guigue)



Cours 8 – lundi 23 octobre 2023

CONTRÔLE DE MI-SEMESTRE

Rappels : contrôle de mi-semestre (voir le site Moodle de l'UE)

- Lundi 6 novembre 2023, de 18h15 à 19h45
- Programme
 - ◆ Tout ce qui aura été vu jusqu'à la semaine 7 incluse, c'est-à-dire les cours 1 à 7 et les semaines de TDTME 1 à 7
- Aucun document autorisé

A savoir

Une question tirée d'un des quizzes pourrait très bien être posée dans le contrôle de mi-semestre...

La semaine des contrôles (du 6 au 10 novembre) il n'y aura pas de cours, TD et TME en LU2IN002

⇒ Prochain cours lundi 13 novembre

CERTAINES SITUATIONS POSENT PROBLÈMES

- Je fais du dessin... Et je veux pouvoir sauver le résultat
- Je fais du dessin... Et je veux afficher facilement le résultat à l'écran

Rappel : en Java, une classe ne peut hériter que d'une **seule** classe
⇒ Limite de l'héritage : pas d'héritage multiple en JAVA
⇒ Mais un autre outil : **les interfaces** va nous permettre de gérer ce genre de problème

PROGRAMME DU JOUR

1 Héritage et interfaces

- Syntaxe et usage des interfaces
- Représentation UML des interfaces
- Propriétés des interfaces

2 Packages

- Créer un package
- Compiler et exécuter avec des packages
- Niveau de visibilité : package

PROGRAMME

1 Héritage et interfaces

- Syntaxe et usage des interfaces
- Représentation UML des interfaces
- Propriétés des interfaces

2 Packages

INTERFACE : DÉFINITION ET USAGE

Usage

Une interface définit :

- un **cahier des charges** (e.g. Voiture, Circuit...)
- une **propriété** (e.g. Sauvegardable, Clonable...)

Elle donne les fonctions à implémenter et leur signature

Ce que contient une interface

- **des signatures de méthodes** (comme les méthodes abstraites)
- mais :
 - ◆ pas de code
 - ◆ pas d'attribut

⇒ Une interface peut être vue comme une "classe abstraite pure"
(classe abstraite sans attribut ni méthode concrète)

INTERFACE : SYNTAXE

- Déclaration d'une interface I contenant une méthode

```
1 public interface I {  
2     public void maMethode(); // méthode abstract  
3 }
```

- Déclaration d'une classe A qui **implémente** l'interface I

◆ Comme pour les classes qui héritent d'une classe abstraite, la classe A doit définir le code de la méthode abstraite

```
4 public class A implements I {  
5     public void maMethode() {  
6         // code  
7     }  
8     ...  
9 }
```

- Une interface ne peut pas être instanciée

```
10 new I(); ERREUR compilation
```

- Subsommption : un objet de type A est aussi de type I

```
11 I ia=new A(); // OK  
12 ia.maMethode(); // OK  
13 ia.methodeDeA(); ERREUR compilation : variable de type I n'est pas un A  
14 ((A)ia).methodeDeA(); // OK
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

7/25

INTERFACE : EXEMPLES D'USAGE (2)

Les interfaces pour énoncer des **propriétés** pour des objets

Exemple : la propriété d'être sauvegardable dans un fichier

★ Qu'est ce qu'un objet Sauvegardable ?

- c'est un objet qui peut être sauvegardé sur disque
- c'est un objet qui répond à la méthode suivante :
 - ◆ void save(String filename)
- On spécifie donc une **interface** précisant ce comportement :

```
1 public interface Sauvegardable {  
2     public void save(String filename);  
3 }
```

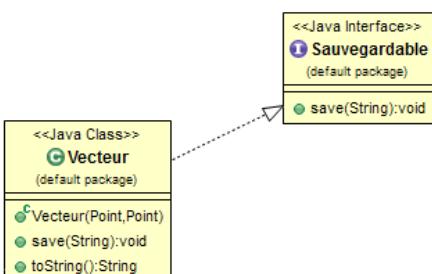
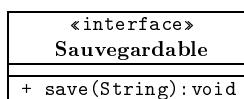
- Toutes les classes qui veulent avoir cette propriété devront implémenter cette interface

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

9/25

INTERFACE : PRÉSENTATION UML

- Une interface est représentée en UML par un rectangle (comme les classes)
- Mais en écrivant au dessus du nom de l'interface : «**interface**»
- La relation "une classe implémente une interface" est représentée par une **ligne en pointillé** avec un **triangle** au bout



©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

11/25

INTERFACE : EXEMPLES D'USAGE (1)

Un **cahier des charges** à respecter

★ Vu de l'extérieur de l'objet, que doit faire un véhicule ?

- accélérer, freiner, tourner
- observation (position, direction, vitesse)

```
1 public interface Véhicule {  
2     // pour le pilotage  
3     public void accelerer(double d);  
4     public void freiner(double d);  
5     public void tourner(double d);  
6  
7     // pour l'observation  
8     public double getVitesse();  
9     public Vecteur getPosition();  
10    public Vecteur getDirection();  
11 }
```

△ Que des méthodes abstraites (**abstract** n'est pas écrit) !!!

- Une classe Voiture qui implémente l'interface Véhicule doit définir le code de toutes ces méthodes

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

8/25

INTERFACE : EXEMPLES D'USAGE (2)

Exemple : la classe **Vecteur**

- Un vecteur est un objet sauvegardable
- ⇒ La classe Vecteur doit **implémenter** l'interface Sauvegardable

```
1 public class Vecteur implements Sauvegardable{  
2     ...  
3     public void save(String filename){  
4         ... // instructions à réaliser  
5     }  
6     ...  
7 }
```

- Comme pour les classes abstraites, Vecteur **doit** contenir le code de la méthode (**contrat**) :
 - ◆ public void save(String filename)

- On est donc sûr de pouvoir sauver un objet Vecteur, il y a donc bien une logique de cahier des charges

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

10/25

INTERFACE & HÉRITAGE : PROPRIÉTÉS (1/3)

Une classe :

- ne peut hériter que **d'une seule classe**
- mais peut implémenter **plusieurs interfaces**

Exemple : dans un logiciel de géométrie, on peut imaginer plusieurs propriétés : dessinable, déplaçable...

```
1 public interface Dessinable {  
2     public void draw();  
3 }  
4 public interface Deplacable {  
5     public void move(double x, double y);  
6 }  
7 public class Polygone extends Figure implements Dessinable, Deplacable {  
8     ...  
9     public void draw() {  
10        // code  
11    }  
12    public void move(double x, double y) {  
13        // code  
14    }  
15 }
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

12/25

INTERFACE & HÉRITAGE : PROPRIÉTÉS (2/3)

- Une interface peut ne pas contenir de méthodes
- ```
1 public interface I { }
```
- Si A implémente une interface I quelconque et B hérite de A alors :
    - ◆ inutile d'écrire dans la signature de B que B implémente I
    - ◆ car B implémente I naturellement (par héritage)
- ```
2 public class A implements I { }
3 public class B extends A { }
```

- On peut utiliser instanceof pour déterminer si un objet implémente une interface

```
4 A a = new A();
5 ab = new B();
6 ib = new B();
7 a instanceof I // true
8 ab instanceof I // true
9 ib instanceof I // true
10 String s = "Bonjour";
11 s instanceof I // false
12
13 a instanceof B // false
14 ab instanceof B // true
15 ib instanceof A // true
```

©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

13/25

APPLICATION

- Il est possible de déclarer une variable d'un type interface
 - ◆ Mais jamais d'instanciation d'une interface
 - ◆ Seules les méthodes de l'interface (et d'Object) sont accessibles
- Application classique : tableau d'un type interface
- Exemple : soit des classes qui implémentent Sauvegardable :
 - ◆ classes Vecteur, Point, Figure,...
 - ◆ classes Personne, Menagerie,...

```
1 Sauvegardable[] tab = new Sauvegardable[3]; //tableau type interface
2 tab[0] = new Vecteur();
3 tab[1] = new Point();
4 tab[2] = new Menagerie(12);
5 for (int i=0; i<tab.length; i++)
6   tab[i].save("fichier"+i);
```

- ★ Très pratique pour appliquer un traitement identique (ici save) à un ensemble de classes qui n'ont rien à voir entre elles...
- ★ Une interface peut être implementée par de nombreuses classes très différentes

©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

15/25

PROGRAMME

1 Héritage et interfaces

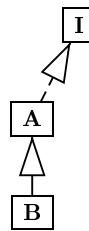
2 Packages

- Créer un package
- Compiler et exécuter avec des packages
- Niveau de visibilité : package

INTERFACE & HÉRITAGE : PROPRIÉTÉS (3/3)

- Si I contient la signature d'une méthode que A ne peut pas implémenter alors (comme pour classes abstraites) :

- ◆ A doit être déclarée abstraite
- ◆ une classe descendante de A devra définir le code de la méthode



```
1 public interface I {
2   public void methodeDeI();
3 }
4 public abstract class A implements I {
5 }
6 public class B extends A {
7   public void methodeDeI() {
8     // code
9   }
10 ...
11 }
```

- On peut utiliser les méthodes de Object à partir d'une variable du type d'une interface

```
12 I ib=new B(); // variable de type I
13 ib.toString(); // OK
14 ib.getClass(); // OK
15 ib.methodeDeI(); // OK
16 ib.methodeDeB(); // Erreur compilation
17 ((B)ib).methodeDeB(); // OK
```

©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

14/25

INTERFACE QUI HÉRITE D'UNE INTERFACE

- Une interface peut hériter d'une autre interface
 - △ C'est un héritage et non pas une implémentation

```
1 public interface Positionnable{
2   public Vecteur getPosition();
3 }
4
5 public interface Deplacable extends Positionnable{
6   public void move(Vecteur v);
7 }
```

- Une classe qui implémente Deplacable :
 - ◆ doit fournir une définition pour la fonction move
 - ◆ doit aussi fournir une définition pour la fonction getPosition

Une remarque avant de finir sur les interfaces

Hors programme : depuis Java 8, on peut écrire dans une interface :

- des attributs public static final
- des méthodes static
- des méthodes default

©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

16/25

INTRODUCTION

Bonne architecture = beaucoup de petites classes...
... chacune étant ciblée, lisible, ré-utilisable
⇒ Le répertoire de projet devient rapidement illisible !

Solution = arborescence de répertoires

- Sous-répertoires associés aux concepts de bas niveaux
- Sous-répertoires associés aux classes utilitaires
- Sous-sous-répertoires de test

Création de packages de classes

PACKAGE

Package java

- Un package est un ensemble de classes mises dans un même répertoire

- Pour définir un package

```
package nomdupackage; // au début du fichier
```

- Pour importer une classe d'un package

```
import nomdupackage.MaClasse;
```

- Pour importer toutes les classes d'un package

```
import nomdupackage.*;
```

Convention d'écritures

Les noms des packages s'écrivent tout en minuscules

Exemples : java.lang, java.util, ...

CRÉER UN SOUS-PACKAGE

Exemple (suite) : on veut créer une classe TestA dans un sous-package `souspaquet1` pour tester la classe A

- dans le fichier `TestA.java` du répertoire `souspaquet1`

```
1 package paquet1.souspaquet1; // création du sous-package
2
3 import paquet1.A; // importation de la classe A pour pouvoir l'utiliser dans TestA
4
5 public class TestA {
6     public static void main(String [] args) {
7         A a1=new A();
8     }
9 }
```

△ Le répertoire `souspaquet1` se trouve dans le répertoire `paquet1`

△ Le nom du package est : `paquet1.souspaquet1`

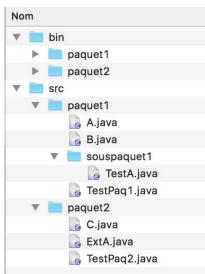
△ Le nom complet de la classe est : `paquet1.souspaquet1.TestA`

COMPILATION / EXÉCUTION DU CODE

- Compilation (position = racine)

- ◆ Spécification d'un répertoire cible : `-d`
- ◆ Spécification du répertoire de gestion des sources : `-cp`

```
> javac -cp src -d bin src/paquet1/souspaquet1/TestA.java
=> Compile l'exécutable + toutes les dépendances (ici A mais pas B ni les autres classes)
```



Exécution

- ◆ Instruction pour se positionner dans le répertoire d'exécution : `-cp`
- ◆ Chemin avec des `.` (pas des `/`)

```
> java -cp bin paquet1.souspaquet1.TestA
```

CRÉER UN PACKAGE

Package java

- Règle java : nom du répertoire = nom du package
- Chaque classe précise le package dans lequel elle est

Exemple : on veut définir un package `paquet1` avec 2 classes A et B et un autre package `paquet2` avec une classe C

- Dans le répertoire `paquet1`

```
1 // Fichier A.java
2 package paquet1;
3
4 public class A { ... }
```

- Dans le répertoire `paquet2`

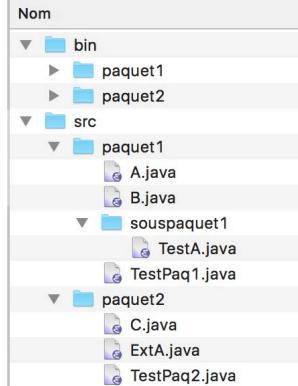
```
1 // Fichier C.java
2 package paquet2;
3
4 public class C { ... }
```

△ Une classe ne peut appartenir qu'à un seul package

DÉCLARATIONS OBLIGATOIRES

Arborescence :

bin/ : les fichiers .class
src/ : les fichiers sources .java



1 Déclaration de paquet

```
1 // Fichier A.java
2 package paquet1;
3 public class A {
4     ...
5 }
```

2 Sous-package

```
1 package paquet1.souspaquet1;
2 import paquet1.A;
3 public class TestA {
4     public static void main(String [] args) {
5         // tests spécifiques à A
6     }
7 }
```

3 Autre package

```
1 package paquet2;
2 import paquet1.A;
3 public class ExtA extends A { // classe fille
4     public ExtA() {
5         super();
6     }
7 }
```

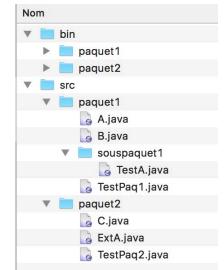
4 Classe du JDK

```
1 import java.util.ArrayList;
```

NIVEAUX DE VISIBILITÉ : PACKAGE

introduction des packages = subtilités sur la visibilité

```
1 package paquet1;
2 public class A {
3     public int i; // public
4     protected int j; // protected
5     private int k; // private
6     int n; // package (nouveau)
7
8     public A(){
9         i=1; j=2; k=3; n=4;
10    }
11 }
```



Visibilités des attributs de A depuis :

	i	j	k	n
Même package	B, TestPaq1	✓	✓	✗
Classe fille	ExtA	✓	✓	✗
Autres cas	C, TestPaq2, TestA	✓	✗	✗

ORGANISATION EN PACKAGES

★ Regrouper les classes en fonction de leur fonctionnalité

Exemple : gestion d'une course de voitures autonomes

1 Réfléchir à un découpage de bas niveau :

- ♦ Circuit
- ♦ Voiture
- ♦ Autonome ⇒ gestion de l'**IA / stratégies**

2 Ajouter les outils (transverses)

- ♦ Gestion de la **géométrie**
- ♦ Gestion des fichiers (sauvegardes/chargements)
- ♦ Interface graphique (IHM)

3 Package de test :

⇒ **sous-répertoire de test** dans chaque package principal

Idée d'utiliser des packages de test

valider le fonctionnement de chaque objet indépendamment
du reste du projet (dans la mesure du possible)

LU2IN002 - Introduction à la programmation orientée objet

Responsable de l'UE et du cours du vendredi :
Christophe Marsala
(e-mail : Christophe.Marsala@lip6.fr)

Cours du lundi : Sabrina Tollari
(e-mail : Sabrina.Tollari@lip6.fr)

(support réalisé à partir de ceux de Christophe Marsala et de Vincent Guigue)



Cours 9 – lundi 13 novembre 2023

PROGRAMME

1 Guide de survie Java

- Entrée et sorties standards
- La classe Scanner

2 Exceptions

PROGRAMME DU JOUR

1 Guide de survie Java

- Entrée et sorties standards
- La classe Scanner

2 Exceptions

- Comment gérer les erreurs à l'exécution ?
- Qu'est-ce qu'une exception ?
- throw : levée d'une exception
- Définir ses exceptions personnalisées
- try ... catch : capture d'exceptions
- finally
- throws
- RuntimeException

ENTRÉE ET SORTIES STANDARDS

La classe System contient 3 attributs public static final

- **System.out** : la sortie standard
 - En général, le terminal
 - ★ Permet de faire : System.out.println(...)
- **System.err** : la sortie erreur standard
 - En général, le terminal
 - ★ Permet de faire : System.err.println(...)
- **System.in** : l'entrée standard
 - En général, le clavier

Remarque : elles correspondent à l'entrée standard et aux sorties standards définies dans l'environnement

SAISIE DE DONNÉES AU CLAVIER

- ★ Comment lire les données saisies au clavier par l'utilisateur ?
■ Utiliser la classe **Scanner** du package **java.util**

```
1 import java.util.Scanner;
2 ...
3 Scanner scan = new Scanner(System.in); // Entrée standart
4 System.out.println("Entrer un entier :"); Entrer un entier :
5 int x = scan.nextInt(); 123
6 System.out.println("x=" + x);
7 System.out.println("Entrer une chaine :"); Entrer une chaine :
8 String s = scan.nextLine(); Bonjour
9 System.out.println("s=" + s); s=Bonjour
```

Remarques :

- Les saisies au clavier ne sont validées qu'après un retour chariot
- La classe Scanner contient plusieurs méthodes pour lire des données en fonction de leur type ou de leur format. Exemples : nextDouble(), nextLine(), next(String format)...

D'autres informations sur Scanner dans les prochains cours...



©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

4/45

EXEMPLE D'UTILISATION DE Scanner (1/2)

Exemple : on veut réaliser un programme qui demande à l'utilisateur de saisir au clavier un choix dans un menu, puis de réaliser certaines opérations en fonction de ce choix

```
1 import java.util.Scanner;
2 ...
3 public class TestScannerMenu {
4     private static final Scanner scan = new Scanner(System.in);
5
6     public static int choixMenu() {
7         System.out.println("____ Menu ____");
8         System.out.println(" 0 - Fin programme");
9         System.out.println(" 1 - Saisir un réel");
10        System.out.println(" 2 - Saisir un booléen");
11        System.out.print("Entrer un choix : ");
12        return scan.nextInt();
13    }
14    public static void main(String [] args) {
15        int choix = choixMenu();
16        while(choix!=0) {
17            switch(choix) {
18                case 1 :
19                    ...
19
```



©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

6/45

EXEMPLE D'UTILISATION DE Scanner (2/2)

```
14 // ... suite
15 int choix = choixMenu();
16 while(choix!=0) {
17     switch(choix) {
18         case 1 :
19             System.out.print("Entrer un réel : ");
20             double reel = scan.nextDouble();
21             System.out.println("reel="+reel);
22             break;
23         case 2 :
24             System.out.print("Entrer un booléen: ");
25             boolean b = scan.nextBoolean();
26             System.out.println("b="+b);
27             break;
28     }
29     choix = choixMenu();
30 }
31 System.out.println("Fin programme");
```

____ Menu ____
0 – Fin programme
1 – Saisir un réel
2 – Saisir un booléen
Entrer un choix : 2
Entrer un booléen : true
____ Menu ____
0 – Fin programme
1 – Saisir un réel
2 – Saisir un booléen
Entrer un choix : 1
Entrer un réel : 3.5
reel=3.5
____ Menu ____
0 – Fin programme
1 – Saisir un réel
2 – Saisir un booléen
Entrer un choix : 0
Fin programme

PROGRAMME

1 Guide de survie Java

2 Exceptions

- Comment gérer les erreurs à l'exécution ?
- Qu'est-ce qu'une exception ?
- throw : levée d'une exception
- Définir ses exceptions personnalisées
- try ... catch : capture d'exceptions
- finally
- throws
- RuntimeException

RAPPELS : ERREURS USUELLES À L'EXÉCUTION

- ★ Certaines erreurs à l'exécution (JVM) sont dues à des erreurs de programmation

Solution : le programmeur lit les messages d'erreurs et corrige

■ NullPointerException

```
1 Point p = null;
2 p.move(1, 0);
3 // Exception in thread "main" java.lang.NullPointerException
4 // at cours1.TestPoint.main(TestPoint.java:2)
```

- ◆ erreur qui arrive souvent dans des cas plus complexes de composition d'objet

■ IndexOutOfBoundsException

```
1 int [] tab = new int [3];
2 tab[5] = 10;
3 // Exception in thread "main"
4 // java.lang.ArrayIndexOutOfBoundsException: 5
5 // at cours1.TestPoint.main(TestPoint.java:2)
```

- ◆ vérifier la ligne et l'index !
- ◆ arrive souvent dans les boucles for

EXCEPTIONS

Du point de vue programmeur :

■ Une exception est une rupture de calcul

- ... qui arrête le programme quand il y a des erreurs
 - ◆ division par zéro
 - ◆ accès à la référence null
 - ◆ ouverture d'un fichier inexistant
 - ◆ ...
- sauf si l'exception est gérée

En Java :

- une exception est un objet
- des mécanismes permettent de gérer l'exception
 - ◆ throw : levée de l'exception
 - ◆ throws : information de propagation de l'exception
 - ◆ try ... catch ... finally : capture de l'exception

AUTRES ERREURS À L'EXÉCUTION

- ★ Certaines erreurs à l'exécution (JVM) ne sont pas dues à des erreurs de programmation, mais aux conditions d'exécution
- Exemple : l'utilisateur saisie une mauvaise valeur
Exemple : l'utilisateur demande de dépiler une pile vide

Que faire ?

- utiliser une valeur spéciale : null, NaN, ...

```
1 double valUtilisateur=2.0;
2 double r = Math.asin(valUtilisateur);
3 // NB : asin n'accepte que des valeurs entre -1 et 1
4 System.out.println(r); // NaN
```

- effectuer une rupture de calcul

```
1 ArrayList<Double> arr = new ArrayList<Double>();
2 arr.add(1.5);
3 arr.add(2.5);
4 int valUtilisateur=5;
5 Double x=arr.get(valUtilisateur);
6 // Exception in thread "main"
7 // java.lang.IndexOutOfBoundsException: Index: 5, Size: 2
8 // at java.util.ArrayList.RangeCheck(ArrayList.java:547)
9 // at java.util.ArrayList.get(ArrayList.java:322)
10 // at Test.main(Test.java:3)
```

EXEMPLE INTRODUCTIF : DÉPILER UNE PILE VIDE

- définition d'un nouveau type d'exception

```
1 public class PileException extends Exception {
2     public PileException(String message) {
3         super("Problème : " + message);
4     }
5 }
```

- levée d'une exception si on essaye de dépiler une pile vide

```
6 public class Pile {
7     ...
8     public int depiler() throws PileException {
9         if (pileVide())
10             throw new PileException("pile vide"); //nouvel objet
11     ...
12 }
```

- gestion de l'exception pour éviter que le programme s'arrête si l'utilisateur demande de dépiler la pile quand elle est vide

```
13 Pile p = new Pile();
14 try {
15     int y = p.depiler(); // lève l'exception si la pile est vide
16     System.out.println(y+" est déplié");
17 } catch (PileException e) {
18     System.out.println(e.getMessage()); // Problème : pile vide
19 }
```

EXCEPTIONS : CRÉATION ET DÉCLENCHEMENT

Une exception est un **objet** d'une classe qui **hérite de la classe Exception** (classe standard du package java.lang)

- Définition d'un nouveau type d'exception

```
1 public class PileException extends Exception {  
2     public PileException(String message) {  
3         super("Problème : " + message);  
4     }  
5 }
```

- Création d'un objet de type Exception

```
PileException e = new PileException("pile vide");
```

- Déclenchement : levée d'une exception (**throw**)

```
throw e;
```

★ Note : création et déclenchement sont souvent combinés

```
throw new PileException("pile vide");
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

13/45

DÉFINIR SES EXCEPTIONS PERSONNALISÉES

On peut **définir ses propres exceptions** en écrivant des classes qui héritent de la classe **Exception** (ou de ses descendantes)

Exemple : définir des exceptions pour gérer les problèmes de pile

```
1 // Exception de base pour tous les problèmes de pile  
2 public class PileException extends Exception {  
3     public PileException(String message) {  
4         super("Problème : " + message);  
5     }  
6 }
```

C'est une classe normale, on peut définir des attributs et méthodes pour transporter des informations sur l'exception, redéfinir **toString** et **getMessage**, et **définir une hiérarchie d'exceptions** :

```
7 // Exception spécifique pour la pile pleine  
8 public class PilePleineException extends PileException {  
9     public PilePleineException() {  
10        super("pile pleine");  
11    }  
12 }  
13 PilePleineException ppe=new PilePleineException();  
14 System.out.println(ppe.getMessage()); //Problème : pile pleine"
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

15/45

CAPTURE D'EXCEPTIONS : try ... catch

- Mise **sous surveillance** d'un ensemble d'instructions qui peuvent déclencher une ou plusieurs exceptions
 - ◆ blocs : **try ... catch**
- **Idée** : si l'exécution de ces instructions produit une exception connue alors :
 - ◆ capturer l'exception
 - ◆ proposer un **traitement approprié**

Syntaxe de base

```
1 try {  
2     // Instructions à exécuter sous surveillance :  
3     // est-ce qu'une exception de type NomException  
4     // est déclenchée ?  
5 } catch (NomException e) { // Si oui, capture de l'exception  
6     // Traitement à effectuer si l'exception a été capturée  
7 }  
8 // Instructions à exécuter une fois le bloc try  
9 // ou le bloc catch terminé
```

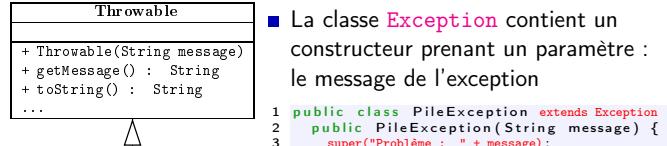
©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

17/45

LES CLASSES STANDARDS Exception ET Throwable

Une exception est un objet d'une classe qui hérite de la classe **Exception** ... qui elle-même hérite de la classe **Throwable**

★ A savoir :



- La classe **Exception** contient un constructeur prenant un paramètre : le message de l'exception

```
1 public class PileException extends Exception {  
2     public PileException(String message) {  
3         super("Problème : " + message);  
4     }  
5 }
```

- Sa classe mère **Throwable** contient :

- ◆ une méthode `getMessage()` qui permet de récupérer le message de l'exception
- ◆ une redéfinition de `toString()` qui retourne : "NomException: message"

```
1 PileException e = new PileException("pile vide");  
2 System.out.println(e.getMessage()); // "Problème : pile vide"  
3 System.out.println(e.toString()); // PileException: Problème : pile vide"
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

14/45

DÉCLENCHEMENT D'EXCEPTIONS

Les instructions qui déclenchent des exceptions :

- sont souvent de la forme :

```
1 if (probleme) {  
2     throw new UneException("il y a un problème !!!");  
3 }  
4 if (pilePleine()) {  
5     throw new PilePleineException();  
6 }
```

mais **△** le mot clef **throw** n'est pas toujours écrit

- cela peut être aussi un appel de méthode

```
7 ArrayList<Double> arr = new ArrayList<Double>();  
8 arr.add(1.5); arr.add(2.5);  
9 Double x=arr.get(5); // déclenche IndexOutOfBoundsException
```

- ou un autre type d'instruction, par exemple, un accès à une case d'un tableau

```
10 int [] tab = new int [3];  
11 tab[5] = 10; // déclenche ArrayIndexOutOfBoundsException
```

★ Comment faire pour éviter que le programme s'arrête ?

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

16/45

CAPTURE D'EXCEPTIONS : EXEMPLE (1/2)

Exemple :

```
1 ArrayList<Double> arr = new ArrayList<Double>();  
2 arr.add(1.5);  
3 arr.add(2.5);  
4 int valUtilisateur=5;  
5  
6 Double x=arr.get(valUtilisateur);  
7 // Exception in thread "main"  
8 // java.lang.IndexOutOfBoundsException: Index: 5, Size: 2
```

★ Comment faire pour qu'au lieu que le programme s'arrête sur une exception, x prenne la dernière valeur de l'ArrayList ?

```
6 Double x=null;  
7 try {  
8     x=arr.get(valUtilisateur);  
9 } catch(IndexOutOfBoundsException e) {  
10     x=arr.get(arr.size()-1);  
11 }  
12 // Qu'il y ait une exception ou pas, le programme continue
```

Ajoutons maintenant temporairement des affichages pour voir les instructions réalisées en fonction de la valeur saisie par l'utilisateur...

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

18/45

CAPTURE D'EXCEPTIONS : EXEMPLE (2/2)

```

6 Double x=null;                                // arr={1.5, 2.5}
7 {
8     System.out.println("avant get");
9     x=arr.get(valUtilisateur);
10    System.out.println("après get");
11 } catch(IndexOutOfBoundsException e) {
12     System.out.println("dans le catch");
13     x=arr.get(arr.size()-1);
14 }
15 System.out.println("Le programme continue avec x="+x);

```

★ Quel est l'affichage obtenu en fonction de valUtilisateur ?

- si valUtilisateur=0 : l'exception n'est pas levée
avant get
après get
Le programme continue avec x=1.5

△ le bloc catch n'est pas effectué

- si valUtilisateur=5 : l'exception est levée
avant get
dans le catch
Le programme continue avec x=2.5

△ toutes les instructions dans le bloc try après l'instruction qui lève l'exception ne sont pas exécutées ("après get" pas affiché)

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 19/45

ATTENTION AUX PORTÉES DES VARIABLES

Les blocs limitent la portée des variables

△ Dans le catch : pas d'accès aux variables déclarées dans le try

Problème de compilation :

```

1 try {
2     int i = 7;
3     ...
4 } catch (Exception e) {
5     System.out.println(i); // ne compile pas : i n'existe pas!
6 }

```

Bonne solution :

```

1 int i = 0; // déclaration avant
2 try {
3     i = 7; // initialisation ici
4     ...
5 } catch (Exception e) {
6     System.out.println(i); // OK
7 }

```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

CAPTURE D'UNE EXCEPTION

★ Que référence la variable dans le catch ?

```

PileException e = new PileException("pile vide");
throw e;
équivalent à :
throw new PileException("pile vide");

```

Si la pile est vide, la méthode depiler() crée un objet de type PileException

```

1 public int depiler() throws PileException {
2     if (pileVide())
3         throw new PileException("pile vide"); // nouvel objet
4     ...
5 }

```

Cet objet est accessible à partir de la variable du catch

```

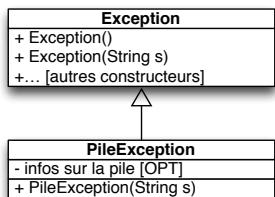
6 try {
7     // dans le cas où la Pile p est vide...
8     p.depiller(); // lève une PileException
9 } catch (PileException e) {
10    System.out.println(e.getMessage()); // "Problème : pile vide"
11 }

```

△ La variable e référence l'objet créé à la ligne 3

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 21/45

CAPTURE ET HIÉRARCHIE D'EXCEPTIONS 1/3



- Possibilité de faire plusieurs catch : traitement séquentiel, le premier qui correspond est utilisé (et les autres non)
- Possibilité de raffiner le traitement en fonction du type de l'exception

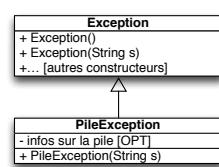
```

1 try {
2     // dans le cas où la Pile p est vide...
3     p.depiller(); // lève une PileException
4
5 } catch (PileException e) { // PileException est attrapée ici
6     System.out.println("Traitement adapté à PileException");
7
8 } catch (Exception e) { // On ne passe pas ici si PileException
9     System.out.println(e.getMessage()); // est attrapée
10 }

```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 23/45

CAPTURE ET HIÉRARCHIE D'EXCEPTIONS 2/3



Une PileException EST UNE Exception...
Le principe de subsumption s'applique

```

Exception e=new PileException("pile vide");

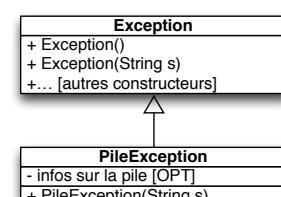
1 try {
2     // dans le cas où la Pile p est vide...
3     p.depiller(); // lève une PileException
4 } catch (Exception e) { // PileException est attrapée
5     System.out.println(e.getMessage());
6 }

```

- PileException est attrapée, car c'EST UNE Exception
- △ catch(Exception e) capture toutes les exceptions

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 22/45

CAPTURE ET HIÉRARCHIE D'EXCEPTIONS 3/3



- On définit obligatoirement les exceptions les plus spécialisées en premier
- Sinon erreur de compilation car code non accessible

Le programme suivant ne compile pas...

```

try {
    // dans le cas où la Pile p est vide...
    p.depiller(); // lève une PileException
}

} catch (Exception e) { // Capture TOUTES les exceptions
    ...
} catch (PileException e) { // Code non accessible
    ...
}

```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 24/45

EXEMPLE : GESTION D'UNE PILE D'ENTIERS (1/5)

Rappel

Une pile est une structure LIFO : Last In First Out.
Le dernier mis dans la pile sort en premier.

1 / Première solution sans gestion des erreurs

```
1 public class Pile {  
2     private int[] items;  
3     private int nbItems; // indice de la 1ère case libre  
4  
5     public Pile() {  
6         items = new int[10];  
7         nbItems = 0;  
8     }  
9     public void empiler(int item) {  
10        items[nbItems++] = item; // syntaxe compacte  
11    }  
12    public int depiler() {  
13        return items[--nbItems];  
14    }  
15    public boolean pilePleine() { return nbItems >= items.length; }  
16    public boolean pileVide() { return nbItems <= 0; }  
17 }
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

25/45

EXEMPLE : GESTION D'UNE PILE D'ENTIERS (3/5)

2 / Deuxième solution avec gestion des erreurs

```
9 public void empiler(int item) throws PilePleineException {  
10    if (pilePleine())  
11        throw new PilePleineException();  
12    items[nbItems++] = item;  
13 }  
14 public int depiler() throws PileException {  
15    if (pileVide())  
16        throw new PileException("pile vide");  
17    return items[--nbItems];  
18 }
```

Erreurs à la compilation :

```
ligne 33: p.empiler(i*10); error: unreported exception PilePleineException; must  
be caught or declared to be thrown  
ligne 36: System.out.println("i=" + i + " " + p.depiller()); error: unreported  
exception PileException; must be caught or declared to be thrown
```

Solution possible : capturer l'exception dans main

★ Où placer le try...catch ?

Solution A : on place le try...catch dans le for

Solution B : on place le try...catch à l'extérieur du for

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

27/45

EXEMPLE : GESTION D'UNE PILE D'ENTIERS (5/5)

Solution B : on place le try...catch à l'extérieur du for

```
34 int valUtilisateur = 5;  
35 try {  
36     for(int i=1; i<= valUtilisateur ; i++)  
37         System.out.println("i=" + i + " " + p.depiller());  
38     } catch (PileException e) {  
39         System.out.println("catch msg=" + e.getMessage());  
40     }  
41 System.out.println("Fin boucle");
```

Quel est l'affichage ?

```
i=1 30  
i=2 20  
i=3 10  
catch msg=Problème : pile vide  
Fin boucle
```

Le try...catch étant à l'extérieur du for, la rupture arrête la boucle for, mais le programme continue

★ Quelle solution choisir ? à l'intérieur ou à l'extérieur ?

Cela dépend de l'application et du résultat que l'on veut obtenir

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

29/45

EXEMPLE : GESTION D'UNE PILE D'ENTIERS (2/5)

Erreur d'utilisation : trop dépiler provoque une erreur

```
31 Pile p = new Pile();  
32 for(int i=1; i<= 3; i++)  
33     p.empiler(i*10); // 10 20 30  
34 int valUtilisateur = 5;  
35 for(int i=1; i<= valUtilisateur ; i++)  
36     System.out.println("i=" + i + " " + p.depiller()); // rupture pour i=4
```

Affichage :

```
i=1 30  
i=2 20  
i=3 10  
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
```

La méthode **depiller** a provoqué une rupture de calcul...

■ Le programme s'est arrêté

◆ Bonne chose !! Si le programme continue avec une exécution faussée, l'erreur est plus dure à déceler

■ Le message n'est pas clair

⇒ il faut détecter l'erreur et envoyer notre message :
interruption garantie + message clair

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

26/45

EXEMPLE : GESTION D'UNE PILE D'ENTIERS (4/5)

Solution A : on place le try...catch dans le for

```
34 int valUtilisateur = 5;  
35 for(int i=1 ; i<= valUtilisateur ; i++) {  
36     try {  
37         System.out.println("i=" + i + " " + p.depiller());  
38     } catch (PileException e) {  
39         System.out.println("catch i=" + i + " msg=" + e.getMessage());  
40     }  
41 }  
42 System.out.println("Fin boucle");
```

Quel est l'affichage ?

```
i=1 30  
i=2 20  
i=3 10  
catch i=4 msg=Problème : pile vide  
catch i=5 msg=Problème : pile vide  
Fin boucle
```

★ Pourquoi 2 affichages pour le catch ?

Il y a eu 2 ruptures pour i=4 et i=5, mais la boucle for a continué ... et les messages d'erreurs sont plus clairs

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

28/45

GESTION D'EXCEPTIONS : VERSION INTÉGRALE

■ Surveiller, gérer, et continuer

◆ blocs : try ... catch ... finally

Définition

Le bloc **finally** contient du code qui sera exécuté **dans tous les cas** (qu'il y ait une exception ou non)

Syntaxe

```
1 try {  
2     ...  
3 } catch (UneException e) {  
4     ...  
5 } catch (UneAutreException e) {  
6     ...  
7 } finally { // TOUJOURS EXÉCUTÉ  
8     ...  
9 }
```

Remarques :

- On peut mettre autant de bloc catch que nécessaire
- Usage du finally : principalement pour la fermeture de fichier/socket

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

30/45

finally : EXEMPLE (1/3)

- Quand il n'y a PAS de levée d'exception

```
34 int valUtilisateur = 2 ; // Dans la pile, toujours 10, 20, 30
35 try {
36     for(int i=1 ; i <= valUtilisateur ; i++) {
37         System.out.println("i=" + i + " " + p.depiler());
38     }
39 } catch (PileException e) {
40     System.out.println("catch msg=" + e.getMessage());
41 } finally {
42     System.out.println("bloc finally toujours exécuté");
43 }
44 System.out.println("Fin");
```

Quel est l'affichage ?

```
i=1 30
i=2 20
bloc finally toujours exécuté
Fin
```

finally : EXEMPLE (3/3)

- Quand il y a levée d'une exception, mais PAS de catch

```
34 int valUtilisateur = 5 ; // Dans la pile, toujours 10, 20, 30
35 try {
36     for(int i=1 ; i <= valUtilisateur ; i++) {
37         System.out.println("i=" + i + " " + p.depiler());
38     }
39 } finally {
40     System.out.println("bloc finally toujours exécuté");
41 }
42 System.out.println("Fin");
```

L'exception¹ est levée pour i=4. Quel est l'affichage ?

```
i=1 30
i=2 20
i=3 10
bloc finally toujours exécuté
Exception in thread "main" PileException: Problème : pile vide
```

△ Le programme s'est arrêté sur une exception, mais le bloc finally a été exécuté avant

¹On suppose qu'on a ajouté throws PileException à la signature de main pour que le programme compile (voir explication slide 36)

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 33/45

throws VERSUS try...catch (1/2)

Si une méthode f appelle une méthode g qui est susceptible de lever une Exception, alors la méthode f doit obligatoirement gérer l'exception levée par g, sinon le code ne compile pas

Exemple : la méthode main appelle la méthode depiler

```
21 public static void main(String [] args) {
22     ...
23     p.depiler(); // peut lever PileException
24 }
```

Erreur à la compilation :

```
error: unreported exception PileException; must be caught or declared to be thrown
p.depiler();
```

Le compilateur nous donne les 2 solutions possibles pour gérer l'exception :

- 1 "must be caught", c-à-d mettre un try...catch pour capturer l'exception dans la méthode
- 2 "or declared to be thrown", c-à-d mettre throws PileException dans la signature de la méthode

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 35/45

finally : EXEMPLE (2/3)

- Quand il y a levée d'une exception et un catch pour la capturer

```
34 int valUtilisateur = 5 ; // Dans la pile, toujours 10, 20, 30
35 try {
36     for(int i=1 ; i <= valUtilisateur ; i++) {
37         System.out.println("i=" + i + " " + p.depiler());
38     }
39 } catch (PileException e) {
40     System.out.println("catch msg=" + e.getMessage());
41 } finally {
42     System.out.println("bloc finally toujours exécuté");
43 }
44 System.out.println("Fin");
```

L'exception est levée pour i=4. Quel est l'affichage ?

```
i=1 30
i=2 20
i=3 10
catch msg=Problème : pile vide
bloc finally toujours exécuté
Fin
```

MOT CLEF throws

Les Exception susceptibles d'être levées dans une méthode doivent être déclarées dans la signature de la méthode avec throws

```
public void maFonction() throws MonException {
    ...
    if (test)
        throw new MonException("MonMessage");
    ...
}
```

Exemple :

```
1 public int depiler() { // on a oublié le throws
2     if (pileVide())
3         throw new PileException("pile vide");
4     return items[--nbItems];
5 }
```

Erreur à la compilation :

```
error: unreported exception PileException; must be caught or declared to be thrown
throw new PileException("pile vide")
```

Solution possible : ajouter throws PileException

```
1 public int depiler() throws PileException { // compilation OK
```

throws VERSUS try...catch (2/2)

- 1 "must be caught", c-à-d mettre un try...catch pour capturer l'exception dans la méthode

```
31 public static void main(String [] args) {
32     ...
33     try {
34         p.depiler();
35     } catch(PileException e) {
36         System.out.println(e.getMessage());
37     }
38 }
```

- 2 "or declared to be thrown", c-à-d mettre throws PileException dans la signature de la méthode

```
41 public static void main(String [] args) throws PileException {
42     ...
43     p.depiler();
44 }
```

★ Quelle solution choisir ?

Cela dépend du résultat que l'on veut obtenir

△ Ne jamais écrire un catch et un throws pour la même exception dans la même méthode, c'est l'un ou l'autre

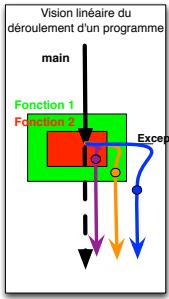
©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 36/45

USAGE : MÉTHODE CAUSANT UNE RUPTURE

Idée (logique de délégation)

Si une méthode Fonction1

- utilise une méthode Fonction2 susceptible de lever MonException
- alors Fonction1 est susceptible de lever MonException également



Résumé des différents cas :

- MonException capturée dans Fonction2 :
 - aucune déclaration nulle part
- MonException capturée dans Fonction1
 - Fonction2 doit déclarer throws MonException
 - Fonction1 ne déclare rien
- MonException capturée dans main
 - Fonction1 et Fonction2 déclarent throws MonException

△ Suivant les cas, on ne reprend pas l'exécution au programme au même endroit...

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

EXEMPLE CORRESPONDANT AU CAS 3

Cas 3 : PileException capturée dans main

- Fonction1 et Fonction2 déclarent throws PileException

```
1 public int depiler() throws PileException { // Fonction2
2     if (pileVide())
3         throw new PileException("pile vide");
4     return items[--nbItems];
5 }
6
7 public void depilerNfois(int n) throws PileException { // Fonction1
8     for (int i=1;i<=n;i++) {
9         System.out.println("i=" + i + " " + depiler());
10    }
11   System.out.println(n + " items dépliés");
12 }
13 catch(PileException e) {
14     System.out.println("depilerNfois : " + e.getMessage());
15 }
16
17 public void main(String [] args) { // main
18     ...
19     p.depilerNfois(5);
20 }
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

EXEMPLE CORRESPONDANT AU CAS 2

Cas 2 : PileException capturée dans Fonction1

- Fonction2 doit déclarer throws PileException
- Fonction1 ne déclare rien

```
1 public int depiler() throws PileException { // Fonction2
2     if (pileVide())
3         throw new PileException("pile vide");
4     return items[--nbItems];
5 }
6
7 public void depilerNfois(int n) { // Fonction1
8     try {
9         for (int i=1;i<=n;i++) {
10            System.out.println("i=" + i + " " + depiler());
11        }
12        System.out.println(n + " items dépliés");
13    } catch(PileException e) {
14        System.out.println("depilerNfois : " + e.getMessage());
15    }
16
17 public void main(String [] args) { // main
18     ...
19     p.depilerNfois(5);
20 }
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

38/45

EXEMPLE CORRESPONDANT AU CAS 1

Cas 1 : PileException capturée dans Fonction2 :

- aucune déclaration nulle part

```
1 public int depiler() { // Fonction2
2     try {
3         if (pileVide())
4             throw new PileException("pile vide");
5         return items[--nbItems];
6     } catch (PileException e) {
7         System.out.println("depiler : " + e.getMessage());
8     }
9     return -1;
10 }
11
12 public void depilerNfois(int n) { // Fonction1
13     for (int i=1;i<=n;i++) {
14         System.out.println("i=" + i + " " + depiler());
15     }
16     System.out.println(n + " items dépliés");
17 }
18
19 public void main(String [] args) { // main
20     ...
21     p.depilerNfois(5);
22 }
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

MOT CLEF throws : CAS PARTICULIER

Attention

Si l'exception est traitée localement, la méthode n'est pas susceptible de la lever : pas de déclaration dans ce cas

```
1 public void maFonction() { // pas de throws
2     ...
3     try {
4         if (test)
5             throw new MonException("MonMessage");
6     ...
7     } catch (MonException e) {
8         ...
9     }
10 }
```

Dans ce cas, il est impossible que maFonction soit interrompue par une MonException non traitée

MOT CLEF throws : REMARQUES

- Une méthode peut lever plusieurs exceptions... et capturer d'autres exceptions

```
1 public void maFonction() throws MonException, MonAutreException {
2     ...
3     if (test1)
4         throw new MonException();
5     ...
6     if (test2)
7         throw new MonAutreException();
8     ...
9     try {
10         ...
11         throw new UneExceptionCapturee();
12     ...
13     } catch (UneExceptionCapturee e) {
14         ...
15         // L'exception est capturée donc pas de throws dans la signature
16     }
17 }
```

△ throws doit être utilisée dans la signature d'une méthode, jamais dans la signature d'une classe

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

41/45

42/45

CASCADE D'EXCEPTIONS

Une Exception peut déclencher d'autres exceptions

Mécanisme : Exception e → catch → Exception e2

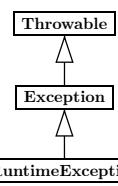
```
1 public void maFonction throws DepassementCapaciteException{  
2     try {  
3         ...  
4  
5     } catch(IndexOutOfBoundsException e) {  
6         ...  
7         throw new DepassementCapaciteException();  
8     }  
9 }
```

- Pour certains cas particuliers

Exception DE TYPE RuntimeException

Rappel : Les **Exception** susceptibles d'être levées dans une méthode doivent être déclarées dans la signature de la méthode avec **throws**

```
public void maFonction() throws MonException { // throws obligatoire  
    ...  
    if(test)  
        throw new MonException("MonMessage");  
    ...  
}
```



Les **RuntimeException** sont des **Exception** particulières qui ne requièrent pas cette déclaration

```
1 public void maFonction() { // pas de throws OK  
2     ...  
3     if(test)  
4         throw new RuntimeException("MonMessage");  
5     ...  
6 }
```

△ La déclaration avec **throws** n'est pas obligatoire, mais il est possible de capturer l'exception avec un **catch**

EXEMPLE : GESTION D'UNE PILE D'ENTIERS

Solution en utilisant des **RuntimeException**

```
1 public void empiler(int item) {  
2     if(pilePleine())  
3         throw new RuntimeException("Pile pleine : ajout impossible");  
4     items[nbItems++] = item;  
5 }  
6  
7 public int depiler() {  
8     if(pileVide())  
9         throw new RuntimeException("Pile vide : extraction impossible");  
10    return items[--nbItems];  
11 }
```

★ Il n'est pas obligatoire de gérer l'exception

Affichage (même code que précédemment slide 26) :

```
i=1 30  
i=2 20  
i=3 10  
Exception in thread "main" java.lang.RuntimeException:  
    Pile vide: extraction impossible
```

- Il y a une rupture de calcul qui provoque l'arrêt du programme
- ... mais le message est plus clair

LU2IN002 - Introduction à la programmation orientée objet

Responsable de l'UE et du cours du vendredi :
Christophe Marsala
(e-mail : Christophe.Marsala@lip6.fr)

Cours du lundi : Sabrina Tollari
(e-mail : Sabrina.Tollari@lip6.fr)

(support réalisé à partir de ceux de Christophe Marsala et de Vincent Guigue)



Cours 10 – lundi 20 novembre 2023

PROGRAMME

1 Flux

- Lecture/écriture dans un flux : principales étapes
- La classe File
- La classe FileInputStream
- La classe DataInputStream
- Lecture/écriture en caractères ou en octets ?
- La classe BufferedReader
- La classe Scanner

2 Fiabilité du code

LES FLUX

Définition

Un **flux** (=stream) : entrées/sorties (dynamiques) d'un programme

Il s'agit d'outils essentiels dès que l'on souhaite :

- Saisir des informations depuis clavier
 - Écrire sur la console
- mais aussi :
- Lire/écrire des fichiers
 - Sauver les paramètres d'un programme (=écrire un fichier!)
 - Communiquer avec d'autres postes de travail (en réseau)
 - Travailler à plusieurs sur un projet (=svn gérer des fichiers)
 - Lire/écrire dans des bases de données (BD)

PROGRAMME DU JOUR

1 Flux

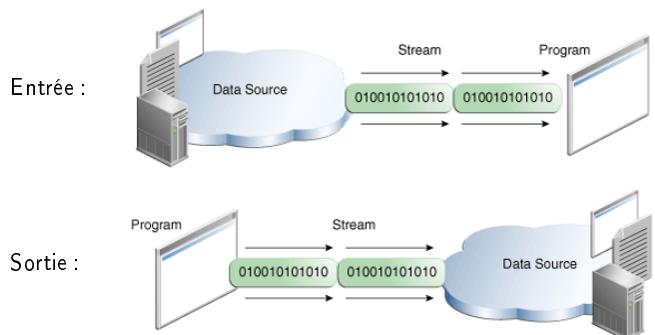
- Lecture/écriture dans un flux : principales étapes
- La classe File
- La classe FileInputStream
- La classe DataInputStream
- Lecture/écriture en caractères ou en octets ?
- La classe BufferedReader
- La classe Scanner

2 Fiabilité du code

- Annotations : @Override...
- Assertions

ENTRÉES/SORTIES D'UN PROGRAMME

Les entrées et sorties sont gérées séparément par flux



JAVA = panoplie d'outils pour communiquer dans les deux sens avec toutes sortes de sources (fichiers, BD, réseaux...)

LES FLUX

★ En Java, le package **java.io** (i pour input et o pour output) contient des classes pour la manipulation des flux

Deux catégories de flux :

- les **flux entrants** pour la lecture
 - ◆ classe InputStream pour lire des octets
 - ◆ classe Reader pour lire des caractères
- les **flux sortants** pour l'écriture
 - ◆ classe OutputStream pour écrire des octets
 - ◆ classe Writer pour écrire des caractères

△ Ces classes sont abstraites

Le nom des classes à utiliser est préfixé par :

- la **source** pour les flux entrants
 - ◆ Exemples : FileInputStream, FileReader, StringReader...
- la **destination** pour les flux sortants
 - ◆ Exemples : FileOutputStream, FileWriter, StringWriter...

PRINCIPALES ÉTAPES

- ★ Quelles sont les principales étapes pour la lecture ou l'écriture dans un flux ?

Exemple : lecture / écriture de données dans un fichier (=flux)

- 1 Vérifier l'existence du fichier, les droits en lecture et/ou écriture...
- 2 Ouvrir (si besoin créer) le fichier en lecture et/ou écriture
- 3 Lire/Écrire des données dans le fichier
 - ◆ Si besoin, vider les buffers
- 4 Fermer le fichier

- ★ D'autres flux se gèrent comme les fichiers : clavier, réseau, BD...

EXEMPLE : LECTURE D'OCTETS DANS UN FICHIER

Etapes 2 et 4 : ouvrir, ... et fermer

- ★ Quelle classe utiliser pour lire des octets dans un fichier ?
- ⇒ La classe `FileInputStream` (car la source est un fichier ⇒ préfixe File et lire des octets ⇒ InputStream)
 - il y a des exceptions à gérer
 - il faut penser à fermer les fichiers ouverts

```
1 File f = new File("tatooine.dat");
2 FileInputStream in = null;
3 try {
4     in = new FileInputStream(f); // ouverture du fichier
5     // throws: FileNotFoundException : => try/catch
6
7     // instructions de LECTURE dans le fichier
8
9 } catch (FileNotFoundException e) {
10 ...
11
12 } finally {
13     if (in != null) {
14         in.close(); // fermeture du fichier
15     }
16 }
```

LES PETITS PIÈGES... LA FERMETURE DES FICHIERS

C Solution plus élégante :

```
1 try {
2     in = new FileInputStream(f);
3     ... // LECTURE
4 } catch (...) {
5 ...
6 } finally{
7     in.close();
8 }
```

D Mais la solution précédente ne marche pas encore... le close est susceptible de lever une exception `NullPointerException` si le fichier n'est pas ouvert !

```
1 try {
2     in = new FileInputStream(f);
3     ... // LECTURE
4 } catch (...) {
5 ...
6 } finally{
7     if (in != null) { // vérifier que le fichier est ouvert
8         in.close();
9     }
10 }
```

LA CLASSE File : PRÉSENTATION DES FICHIERS

Etape 1 : vérifier l'existence du fichier, les droits...

La classe `File` permet de créer un objet représentant un fichier

Nombreuses opérations très intéressantes concernant la manipulation des fichiers :

- test d'existence
- distinction fichier/répertoire
- effacement
- ...

Exemple :

```
1 File f = new File("tatooine.dat");
2 if (f.exists()) {
3 ...
4 }
```

△ `new File(...)` = création d'un objet de la classe `File` qui représente un fichier, ne crée pas un fichier

△ La classe `File` n'est pas un flux

LES PETITS PIÈGES... LA FERMETURE DES FICHIERS

★ Où fermer le fichier ?

A Toujours fermer un fichier qui a été ouvert...

```
1 try {
2     FileInputStream in = new FileInputStream(f);
3     ... // LECTURE
4     in.close();
5 } catch (...) {
6 ...
7 }
```

B ... même s'il y a des erreurs pendant la lecture !

```
1 FileInputStream in = null;
2 try {
3     in = new FileInputStream(f);
4     ... // LECTURE
5     in.close();
6 } catch (...) {
7     in.close();
8 }
```

△ Bien mettre la déclaration en dehors du try, sinon ça ne compilera pas !

LECTURE D'OCTETS DANS UN FICHIER (SUITE)

Etape 3 : lire

La classe abstraite `InputStream` contient une méthode abstraite `read()` pour lire un octet dans un flux.

Elle est redéfinie dans la classe `FileInputStream` :

```
public int read() throws IOException
    ■ Reads a byte of data from this input stream. This method blocks if no input is yet available.
    ■ Returns: the next byte of data, or -1 if the end of the file is reached.
    ■ Throws: IOException - if an I/O error occurs.
```

Exemple d'instructions de LECTURE :

```
1 int c = in.read(); // lecture d'un octet d'information
2           // susceptible de lever IOException => try/catch
3 while (c != -1) { // tant que fin de fichier non atteinte
4     System.out.print(c); // affichage
5     c = in.read(); // lecture du caractère suivant
6 }
```

Ce qui peut aussi s'écrire plus simplement :

```
1 while ((c = in.read()) != -1) {
2     System.out.print(c);
3 }
```

LIMITES DE FILEINPUTSTREAM

- Lecture octet par octet
 - ◆ accès "bas niveau" au fichier
- Rappels Java : représentation interne
 - ◆ entiers: binaire (fort/faible) signé en complément à 2
 - byte : 1 octet
 - short : 2 octets
 - int : 4 octets
 - ◆ réels : norme IEEE 754, signe+exposant+significande
 - float : 4 octets
 - double : 8 octets
 - ◆ char : 2 octets (Unicode)
- Bilan : complexe de reconstruire des valeurs lues par octet
- Solution : nouvelles classes accès "haut niveau" qui contiennent des méthodes utiles

LIMITES → NOUVELLES CLASSES

Des classes supplémentaires enrichissent les classes de base

- Exemple : classe de lecture de haut niveau : `DataInputStream`

```
1 File f = new File("tatoonie.dat");
2 FileInputStream in = null;
3 DataInputStream istream = null;
4 try {
5     in = new FileInputStream(f); // ouverture du fichier
6     // on encapsule le flux dans un objet DataInputStream
7     istream = new DataInputStream(in);
8
9     System.out.println(istream.readInt());
10    System.out.println(istream.readDouble());
11    System.out.println(istream.readFloat());
12    System.out.println(istream.readChar());
13    // mais pas de fonctions pour les String...
14
15 } catch (...) {
16     ... // à compléter ...
17 } finally {
18     if (istream != null)
19         istream.close();
20 }
```

- Il existe une classe équivalente pour les flux de sortie

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 14/34

BILAN : FLUX POUR LIRE DEPUIS UN FICHIER

Lecture du flux (toujours du début à la fin : flux séquentiel) :

- une lecture peut lever une `IOException`
- lecture de bas niveau : octet par octet
 - int c = in.read(); // lecture d'un octet
// si c vaut -1 : fin de fichier atteinte
- lecture de haut niveau :
 - ◆ encapsulation du `FileInputStream` dans `DataInputStream`

```
DataInputStream istream = new DataInputStream(in);
```
 - ◆ méthodes disponibles:
 - istream.readInt(); // rend un int
 - istream.readDouble(); // rend un double
 - istream.readFloat(); // rend un float
 - istream.readChar(); // rend un char
 - ◆ la fermeture du flux peut se faire depuis `DataInputStream`

```
istream.close();
```
 - ◆ si on ne peut pas lire ce qui est demandé : `EOFException`
 - par exemple : un int correspond à 4 octets
 - utile pour détecter la fin du fichier

ÉCRITURE DE BAS NIVEAU

- ★ Même principe pour l'écriture d'octets...

Exemple : quelle classe pour écrire des octets dans un fichier ?

⇒ La classe `FileOutputStream`

Signature d'un des constructeurs

```
public FileOutputStream(File file, boolean append) throws
FileNotFoundException
```

- Signature proche de celle d'ouverture en lecture...

■ ... avec une option supplémentaire append : **ajouter des choses** à la fin du fichier au lieu du début

```
1 File fSortie = new File("dagobah.dat");
2 FileOutputStream out = null;
3 try {
4     out = new FileOutputStream(fSortie, true);
5     ...
6     out.write(c);
7 }
```

△ Par défaut ou si append=false, alors remplacement du fichier existant. Gare aux catastrophes !

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 16/34

ÉCRITURE DE HAUT NIVEAU

★ Quelle classe pour écrire des entiers, réels, caractères... ?

⇒ La classe `DataOutputStream`

```
1 FileInputStream in = null;
2 FileOutputStream out = null;
3 DataOutputStream ostream = null;
4 try {
5     out = new FileOutputStream(fSortie, true);
6     ostream = new DataOutputStream(out);
7     ...
8     ostream.writeInt(1);
9     ostream.writeDouble(1.5);
10    ostream.writeFloat(1.5f);
11    ostream.writeChar('A');
12    ...
13 } catch (...) {
14     ...
15 } finally {
16     if (ostream != null)
17         ostream.close();
18 }
19 }
```

EXEMPLE : ÉCRITURE D'OCTETS DANS UN FICHIER

- Exemple d'utilisation (Oracle Java Tutorials) :

```
1 FileInputStream in = null;
2 FileOutputStream out = null;
3
4 try {
5     in = new FileInputStream("tatoonie.dat");
6     out = new FileOutputStream("dagobah.dat");
7
8     int c = in.read();
9     while (c != -1) {
10         out.write(c);
11         c = in.read();
12     }
13 } catch (...) {
14     ...
15 } finally {
16     if (in != null) {
17         in.close();
18     }
19     if (out != null) {
20         out.close();
21     }
22 }
```

■ Que fait ce programme ?

Recopie dans le fichier dagobah.dat le contenu du fichier tatoonie.dat

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 17/34

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 18/34

CLASSES POUR LES FLUX : BILAN (1)

- Package `java.io`
- Classe `File` : créer un objet pour représenter un fichier
 - ◆ gérer un fichier ou un répertoire (créer, lire,...)
- `Flux` :
 - ◆ flux entrant : `input`
 - classes abstraites : `InputStream` (général), `Reader` (caractères)
 - accès : `FileInputStream` (octet), `FileReader` (caractères)
 - ◆ flux sortant : `output`
 - classes abstraites : `OutputStream` (général), `Writer` (caract.)
 - accès : `FileOutputStream` (octet), `FileWriter` (caractères)
- Accès de haut niveau à un flux
 - ◆ lecture : `DataInputStream`
 - ◆ écriture : `DataOutputStream`

LECTURE/ÉCRITURE EN CARACTÈRES OU OCTETS ?

★ Comment lire/écrire des caractères ?

Différence entre caractères...

- Fichiers de textes
 - Entêtes des fichiers
 - XML, HTML...
 - Parfois pour quelques chiffres
 - ◆ lorsque la précision n'est pas importante
 - ◆ pour les fichiers CSV...
- ... et données stockées en octets
- Fichiers de chiffres
 - ◆ Volume
 - ◆ Précision

LECTURE DE CACRACTÈRES DANS UN FICHIER

- Encore une nouvelle classe : `BufferedReader`...
- ... qui introduit une nouvelle méthode :

```
public String readLine() throws IOException
```

Reads a line of text. A line is considered to be terminated by any one of a line feed or a carriage return.

Returns: A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

Throws: IOException - If an I/O error occurs

```
1 BufferedReader in = null;
2 FileReader fr = null;
3 try {
4     fr = new FileReader("tatooine.txt"); //lecture de caractères non octets
5     in = new BufferedReader(fr);
6
7     String buf = in.readLine();
8     while(buf != null){
9         System.out.println(buf);
10        buf = in.readLine();
11    }
12 }...
```

CLASSES POUR LES FLUX : BILAN (2)

- Package `java.io`
- Classe `File` : créer un objet pour représenter un fichier
 - ◆ gérer un fichier ou un répertoire (créer, lire,...)
- `Flux` :
 - ◆ flux entrant : `input`
 - classes abstraites : `InputStream` (général), `Reader` (caractères)
 - accès : `FileInputStream` (octet), `FileReader` (caractères)
 - ◆ flux sortant : `output`
 - classes abstraites : `OutputStream` (général), `Writer` (caract.)
 - accès : `FileOutputStream` (octet), `FileWriter` (caractères)
- Accès de haut niveau à un flux
 - ◆ lecture par type : `DataInputStream`
 - ◆ écriture par type : `DataOutputStream`
 - ◆ caractères : accès par ligne : `BufferedReader`, `BufferedWriter`
 - combiner avec `String Tokeniser`
 - ◆ caractères : encore plus souple : `Scanner`

STRATÉGIE LECTURE DE FICHIERS EN CARACTÈRES

Une fois la ligne lue, il peut être nécessaire de la traiter...

- Séparer les mots : classe `StringTokenizer` du package `java.util`
 - ◆ choix des séparateurs (espace, tabulation, virgule...), accès aux sous-chaines
- Conversion avec les méthodes des classes enveloppes
 - ◆ Exemple : dans la classe `Double`

```
String buf = "1.5";
double x = Double.valueOf(buf);
```
 - ◆ Méthodes similaires avec les classes `Integer`, `Float`...

STRATÉGIE ALTERNATIVE

Idée : combiner la lecture de fichier avec un `Tokenizer`...

→ la classe `Scanner` du package `java.util`.

Exemples de signatures de constructeurs :

```
public Scanner(InputStream source)
public Scanner(Readable source) //interface
```

```
1 Scanner s = null;
2 try {
3     s = new Scanner(new BufferedReader(new FileReader("text.txt")));
4     while (s.hasNext()) {
5         System.out.println(s.next());
6     }
7 } catch (...) {
8 ...
9 } finally {
10     if (s != null) {
11         s.close();
12     }
13 }
```

Par défaut, le séparateur est un espace mais on peut le changer...

```
1 s.useDelimiter(" ");
```

PROGRAMME

1 Flux

2 Fiabilité du code

- Annotations : `@Override...`
- Assertions

ENJEUX DU COURS

Améliorer la fiabilité des programmes développés

Besoin d'outils pour donner confiance dans le code développé...

- 1 Objectif 1 : code compilé = code fonctionnel
 - ♦ les erreurs de compilation sont les plus faciles à corriger...
 - ♦ respectons les règles de développement
 - ♦ utilisons des outils pour provoquer des erreurs de compilation si le code prend mauvaise tournure...
 - ♦ annotations (aide au compilateur)
- 2 Objectif 2 : en cours d'exécution, détectons les erreurs et informons l'utilisateur
 - ♦ fonctionnement des ruptures (exceptions)
 - ♦ déclenchement, traitement....
- 3 Vers la programmation par contrat (assertions)

FIABILITÉ = RESPECT DES RÈGLES DE DÉVELOPPEMENT

Idée

Pour éviter les erreurs, respecter les règles :

- Choix des noms pour comprendre qui fait quoi
- Classes et méthodes de taille raisonnable, limiter les accès public
 - ♦ les opérations complexes sont déléguées à d'autres classes
 - ♦ le client voit peu de choses : facile à comprendre, évite les failles
 - ♦ taille limitée = on peut envisager de relire le code de la classe si nécessaire
- Evolutivité/architecture réfléchie (pour éviter les modifications ultérieures...), usage de `final...`
- Plus le code est clair, plus les erreurs sont faciles à voir



EXEMPLE SUR LA VOITURE (1)

Implémenter une voiture = 2 visions

- Vision client : accès aux contrôles comme un pilote (pédales, volant)
- Vision développeur : physique complexe à gérer pour obtenir un comportement réaliste

1 Limiter la vision public :

- ♦ commandes pédales/volant ⇒ `public`
- ♦ calcul de la mise à jour de la position/direction, dérapage éventuel ⇒ `private`

2 Limiter la taille des méthodes/classes

- gestion de la physique = moteur physique ou géométrie dans l'espace... ⇒ `classes outils, vecteur, fichier` gérées à part



VÉRIFICATIONS STATIQUES

- ### Idée
- Faire en sorte de vérifier un maximum de chose au niveau de la compilation...
⇒ plus facile à corriger

- Par défaut le compilateur vérifie
 - ♦ `syntaxe` (les ;, parenthèses, accolades...)
 - ♦ `type` des variables et compatibilité avec les instances et méthodes appelées
 - ♦ `niveau d'accès` (aux méthodes, variables...)
- D'autres propriétés sont plus difficiles à montrer et nécessitent plus d'informations transmises au compilateur
 - ♦ `langage d'annotations`

Idée

1 Code morcelé = code plus lisible

2 Code morcelé = code testable

Développer plusieurs fonctions `main` pour tester le bon fonctionnement des différentes fonctions basiques

En séparant la gestion de la physique de la voiture, il devient possible de tester chaque fonction du moteur physique...

- plus facile de tester/corriger des méthodes basiques plutôt que l'ensemble d'une méthode très complexe

NB : prémisses de la programmation par contrat (cf cours L3 "POO avancée")



ANNOTATIONS STANDARDS (1/2)

Certaines erreurs ne posent pas de problème de compilation mais provoquent des comportements étranges lors de l'exécution... Ce sont les plus chères à corriger!

- Exemple : dans la classe Point, on veut redéfinir `toString` :

```
1 public String toString() {  
2     return "Point [x=" + x + ", y=" + y + "]";  
3 }
```

- ◆ Pas d'erreur à la compilation
- ◆ Pas d'exception levée à l'exécution

◆ MAIS problème lors de l'exécution :

```
Point p=new Point();  
System.out.println(p.toString());
```

- ◆ La méthode `toString` de la classe `Object` est utilisé !!!

- ★ Comment demander au compilateur de signaler ce problème ?

ANNOTATIONS STANDARDS (2/2)

Solution : utilisation d'`annotations`

- `@Override` : pour indiquer une redéfinition de méthode

Exemple :

```
1 @Override  
2     public String toString() { // => erreur de compilation  
3         return "Point [x=" + x + ", y=" + y + "]";  
4     }
```

Provoque une erreur de compilation :

```
Point.java: method does not override or implement a method  
from a supertype  
@Override  
^  
1 error
```

Il existe de nombreuses autres annotations :

- `@Deprecated` : pour indiquer un élément "déprécié", engendre un warning

■ ...

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

32/34

VÉRIFICATIONS DYNAMIQUES

Idées

Une fois la compilation passée, il reste de nombreuses erreurs possibles...

Les tests ne sont pas finis!

- ⇒ faire des tests sur les `arguments` des méthodes pour vérifier la faisabilité
- ⇒ faire des tests sur les `sorties` des méthodes pour vérifier la crédibilité des résultats obtenus

Exemples :

- Tester si la case demandée dans un tableau existe avant de retourner le résultat
- Tester si la valeur de retour de la fonction `carre` est bien positive
- Tester si l'argument de la division est bien différent de 0
- ...

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

LES ASSERTIONS

Lors de la phase de développement du programme, le mécanisme des assertions permet au programmeur de vérifier dynamiquement des conditions

- Syntaxe : `assert expr1 [: expr2]`

- ◆ `expr1` est une expression booléenne
- ◆ `expr2` de type quelconque

- Exécution :

- ◆ `expr1` est évaluée
- ◆ si `false` ⇒ une `AssertionError` est lancée
- ◆ `expr2`, convertie en chaîne, est utilisée comme message
 - `java.lang.Object`
 - extended by `java.lang.Throwable`
 - extended by `java.lang.Error`
 - extended by `java.lang.AssertionError`

Alternative avec des exceptions :

```
1 if (! expr1) {throw new MyException();}
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

34/34

LU2IN002 - Introduction à la programmation orientée objet

Responsable de l'UE et du cours du vendredi :
Christophe Marsala
(e-mail : Christophe.Marsala@lip6.fr)

Cours du lundi : Sabrina Tollari
(e-mail : Sabrina.Tollari@lip6.fr)

(support réalisé à partir de ceux de Christophe Marsala et de Vincent Guigue)



Cours 11 – lundi 27 novembre 2023

PROGRAMME

1 Flux (suite)

- La classe RandomAccessFile : lecture et écriture
- String, Console, réseaux, BD...
- Entrée et sorties standards

2 Conteneurs génériques

3 Design Patterns

4 Exercices d'annales : Examen Janvier 2020 (2019oct)

VERS LES BASES DE DONNÉES...

Fichier structuré

Dans certains fichiers, on souhaite stocker des données structurées.
Exemple :

- des fichiers clients avec des colonnes structurées (eg : nom, prénom, date de naissance, rue, code postal, ville)

En général, on souhaite faire des traitements associé et ... **il faut aller assez vite.** Exemple :

- pour accéder à la fiche d'un client, on ne veut pas parcourir tout le fichier

Remarques : dans ce type de fichier structuré, il peut être utile :

1 de connaître la longueur des types usuels

Exemple : int=4 octets, double=8 octets, ...

2 d'imposer des tailles fixes pour les chaînes de caractères

Exemple : les noms doivent être obligatoirement stockés sur 10 caractères : String.format("%10s", "toto")

PROGRAMME DU JOUR

1 Flux (suite)

- La classe RandomAccessFile : lecture et écriture
- String, Console, réseaux, BD...
- Entrée et sorties standards

2 Conteneurs génériques

- Création d'une classe générique
- Extension d'une classe générique
- wildcard

3 Design Patterns

4 Exercices d'annales : Examen Janvier 2020 (2019oct)

- Exercice 1 : Combien d'instances, quelle méthode?

RAPPELS : CLASSES POUR LES FLUX

- Package `java.io`
- Classe `File` : créer un objet pour représenter un fichier
 - ◆ gérer un fichier ou un répertoire (créer, lire,...)
- `Flux` :
 - ◆ flux entrant : `input`
 - classes abstraites : `InputStream` (général), `Reader` (caractères)
 - accès : `FileInputStream` (octet), `FileReader` (caractères)
 - ◆ flux sortant : `output`
 - classes abstraites : `OutputStream` (général), `Writer` (caract.)
 - accès : `FileOutputStream` (octet), `FileWriter` (caractères)
- Accès de haut niveau à un flux
 - ◆ lecture par type : `DataInputStream`
 - ◆ écriture par type : `DataOutputStream`
 - ◆ caractères : accès par ligne : `BufferedReader`, `BufferedWriter`
- ★ Rappel : parcours du flux (du début à la fin : flux séquentiel)

★ Comment lire et écrire dans le même fichier et à n'importe quelle position ?



©2021-2022 C. Marsala / V. Guigue

LU2IN002 - POO en Java

4/52

CLASSE RandomAccessFile (1/3)

La classe `RandomAccessFile` permet d'accéder en lecture et en écriture à n'importe quelle position du fichier

△ Ici "random" ne signifie pas "aléatoire", mais "n'importe quelle position"

■ Création/ouverture assez classique

```
public RandomAccessFile(File file, String mode)
                        throws FileNotFoundException
```

avec mode :

◆ "r" : Open for reading only. Invoking any of the write methods of the resulting object will cause an IOException to be thrown.

◆ "rw" : Open for reading and writing. If the file does not already exist then an attempt will be made to create it.

CLASSE RandomAccessFile (2/3)

- Un fichier à accès aléatoire se comporte comme un très grand tableau d'octets
- Une sorte de curseur (index) appelé "file pointer" indique la position actuelle (*offset*) dans le fichier
- △ La position est en octets toujours depuis le début du fichier

Méthodes utiles :

- Fonction `getFilePointer` retourne la position actuelle

```
public long getFilePointer() throws IOException
    ♦ Returns: the offset from the beginning of the file, in bytes, at which the next read or write occurs.
```

- Fonction `seek` qui déplace le curseur dans le fichier

```
public void seek(long pos) throws IOException
    ♦ pos - the offset position, measured in bytes from the beginning of the file, at which to set the file pointer.
```

EXEMPLE : SAUVER UNE LISTE DE POINTS (1/3)

```
1 public class Point {
2     private static int cpt=0;
3     private final int id;
4     private double x, y;
5     public Point(double x, double y) {
6         cpt++;
7         id=cpt;
8         this.x=x; this.y=y;
9     }
10    public String toString() {
11        return "Point "+id+" ("+x+","+y+")";
12    }
13 }
```

Dans la méthode `main`, une liste de points à sauver :

```
14 ArrayList<Point> listePoints=new ArrayList<Point>();
15 listePoints.add(new Point(1,2));
16 listePoints.add(new Point(3,4));
17 listePoints.add(new Point(5,6));
```

- ★ Comment sauver dans un fichier les données de la liste de points ?

EXEMPLE : SAUVER UNE LISTE DE POINTS (3/3)

- ★ Comment lire les données d'un point dans un flux ?

On ajoute dans la classe `Point` un constructeur qui prend en paramètre un flux avec lequel on peut lire.

```
1 public Point(DataInput flux) throws IOException {
2     id=flux.readInt();
3     x=flux.readDouble();
4     y=flux.readDouble();
5 }
```

Dans une autre méthode `main`, on peut accéder à un point choisi :

```
7 final int TAILLE = 20; // 20 octets pour stocker un point
8 Scanner scan=new Scanner(System.in);
9 System.out.print("Entrer le numéro du point :");
10 int numero = scan.nextInt();
11 RandomAccessFile raf = new RandomAccessFile("pointsRAF.dat","r");
12 raf.seek((numero-1)*TAILLE);
13 Point p=new Point(raf);
14 System.out.println("Lecture du point "+numero+": "+p);
15 raf.close();
```

Affichage :

```
Entrer le numéro du point : 2
Lecture du point 2 : Point 2 (3.0,4.0)
```

△ Ne pas oublier de gérer les exceptions

CLASSE RandomAccessFile (3/3)

La classe `RandomAccessFile` implémente les interfaces :

- `DataInput`¹
- `DataOutput`²

qui contiennent de nombreuses méthodes de haut niveau

	DataInput	DataOutput
Exemples :	int readInt() double readDouble() char readChar() String readUTF() String readLine()	writeInt(int) writeDouble(double) writeChar(int) writeUTF(String)

¹implémentée aussi par `DataInputStream`

²implémentée aussi par `DataOutputStream`

EXEMPLE : SAUVER UNE LISTE DE POINTS (2/3)

- ★ Comment écrire les données d'un point dans un flux ?

Dans la classe `Point`, on ajoute une méthode `écrire` qui prend en paramètre un flux avec lequel on peut écrire.

```
1 public void écrire(DataOutput flux) throws IOException {
2     flux.writeInt(id);
3     flux.writeDouble(x);
4     flux.writeDouble(y);
5 }
```

Dans la méthode `main` :

```
6 RandomAccessFile raf = new RandomAccessFile("pointsRAF.dat","rw");
7 for(Point p : listePoints) {
8     System.out.println("Écriture "+p+" position="+raf.getFilePointer());
9     p.écrire(raf);
10 }
11 System.out.println("A la fin position="+raf.getFilePointer());
12 raf.close();
```

Affichage :

```
Écriture Point 1 (1.0,2.0) position=0
Écriture Point 2 (3.0,4.0) position=20
Écriture Point 3 (5.0,6.0) position=40
A la fin position=60
```

On voit que chaque point est stocké sur 20 octets
■ 1 int = 4 octets
■ 2 double = 2x 8 octets

COMPARAISON AVEC InputStream/OutputStream

- Écriture avec `OutputStream` :

```
1 FileOutputStream fos=new FileOutputStream("pointsOS.dat");
2 DataOutputStream dos=new DataOutputStream(fos);
3 for(Point p : listePoints) {
4     p.écrire(dos);
5 }
6 dos.close();
```

Lecture avec `InputStream` (on est obligé de parcourir le fichier) :

```
7 FileInputStream fis=new FileInputStream("pointsOS.dat");
8 DataInputStream dis=new DataInputStream(fis);
9 boolean onContinue=true;
10 int cpt=0;
11 while( onContinue ) {
12     try {
13         Point p=new Point(dis);
14         System.out.println("Lecture du "+p);
15         cpt++;
16     } catch (EOFException e) {
17         onContinue = false; // On a atteint la fin du fichier
18     }
19 }
20 System.out.println(cpt+" points ont été lus");
21 dis.close();
```

LES FLUX DÉPASSENT LE CADRE DES FICHIERS

Les flux dépassent le cadre des fichiers :

- String, Console, réseaux, BD...

Exemple :

- Lire une String comme un flux avec la classe `StringReader`

```
1 String s = "blablabla ...";
2
3 // initialisation à partir d'une String
4 StringReader sread = new StringReader(s);
5
6 System.out.println(sread.read()); // -> 98 (code de 'b')
7 System.out.println(sread.read()); // -> 108 (code de 'l')
8 System.out.println(sread.read()); // -> 97 (code de 'a')
```

- Fonctionnalité très utile pour unifier une chaîne de traitements
 - ◆ tous les types d'entrées peuvent être des flux, on peut jouer avec l'héritage

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 13/52

LECTURE HTTP SUR LE WEB

- Lire sur internet est une simple formalité...

- ◆ outil pour les URL → flux
 - ◆ ouverture / lecture classique !

```
1 import java.net.URL ;
2 ...
3 URL url = new URL("http://www.yahoo.fr");
4 InputStream is = url.openStream();
5 InputStreamReader isr = new InputStreamReader(is, "utf8") ;
6 BufferedReader bf=new BufferedReader(isr);
7 String buf = bf.readLine();
8 while(buf != null){
9     buf = bf.readLine();
10    System.out.println(buf);
11 }
12 bf.close();
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 15/52

ENTRÉE ET SORTIES STANDARDS

La classe `System` contient 3 attributs déclarés `public static final` qui sont des flux. Ils correspondent à l'entrée standard et aux sorties standards définies dans l'environnement.

- `System.out` de type `PrintStream` : la sortie standard
 - En général, le terminal
 - ★ Permet de faire : `System.out.println(...)`
- `System.err` de type `PrintStream` : la sortie erreur standard
 - En général, le terminal
 - ★ Permet de faire : `System.err.println(...)`
- `System.in` de type `InputStream` : l'entrée standard
 - En général, le clavier

La classe `PrintStream` hérite de `OutputStream` et contient des surcharges de méthodes `print`, `println`, `format...`

OUTIL AVANCÉ POUR LA CONSOLE

- Classe `Console` du package `java.io`

- ◆ Permet d'accéder à la console associée à la machine virtuelle Java actuelle (si il y en a une)

```
1 Console c = System.console();
2 if (c == null) {
3     System.err.println("No console.");
4     System.exit(1);
5 }
6
7 String login = c.readLine("Enter your login: ");
8 char [] oldPassword = c.readPassword("Enter your old password: ");
```

Enter your login: toto

Enter your old password:

- Notons la forme particulière de la construction de la console
- Console est en réalité un Singleton
 - ◆ L'unique constructeur de la classe `Console` est privé

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 14/52

VERS LES BD, SQL

- Connection aux bases de données aisée

```
1 // le driver gère la connection (à MySQL par exemple)
2 Connection con = DriverManager.getConnection(
3         "jdbc:mysql:myDriver:myDatabase",
4         username,
5         password);
6
7 // instantiation de la connection
8 Statement stmt = con.createStatement();
9
10 // envoi d'une requête
11 ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
12
13 // traitement séquentiel des réponses
14 while (rs.next()) {
15     int x = rs.getInt("a");
16     String s = rs.getString("b");
17     float f = rs.getFloat("c");
18 }
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 16/52

PROGRAMME

1 Flux (suite)

2 Conteneurs génériques

- Création d'une classe générique
- Extension d'une classe générique
- `wildcard`

3 Design Patterns

4 Exercices d'annales : Examen Janvier 2020 (2019oct)

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java 17/52

PROBLÉMATIQUE GÉNÉRALE (1/2)

Conteneurs génériques

- Construire une structure de données adaptée à différents types d'entrées. Exemples :
 - Liste d'**entiers**, liste de **réels**, liste de **String**, liste de **Point**...
 - ⇒ ne pas construire une classe pour chaque cas !!!

Solution (avant Java 1.5) :

```
1 public class ListeGenOld {  
2     private final static int TAILLE_MAX = 500;  
3     private Object[] liste;  
4     private int size;  
5     public ListeGenOld(){  
6         liste = new Object[TAILLE_MAX];  
7         size = 0;  
8     }  
9     public void add(Object o){ liste[size] = o; size++;}  
10    public Object get(int i){return liste[i];}  
11    ...  
12 }  
13 // main : une liste de Point ?  
14 ListeGenOld liste = new ListeGenOld();  
15 liste.add(new Point()); // OK  
16 Point p = (Point)liste.get(0); // cast obligatoire  
17 liste.add("Bonjour"); // OK !!!
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

19/52

CONTENEURS GÉNÉRIQUES

Le package `java.util` propose de nombreux conteneurs génériques. Exemples :

- HashSet, HashMap
- ArrayList, ArrayDeque
- TreeMap, TreeSet
- LinkedList
- LinkedHashSet, LinkedHashMap

General-purpose Implementations					
Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementations
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

21/52

... ET USAGE CLIENT

La syntaxe est celle des `ArrayList`... Vous la connaissez déjà!

```
1 Paire<Integer , String> p1 = new Paire<Integer , String >(2, "toto");  
2 Integer x=p1.getEl1();  
3 String s1=p1.getEl2();
```

- Le type est donc : `Paire<Integer, String>`
- Le type du contenu est passé en argument *spécial* entre `<>`

On peut utiliser la même classe paire, mais avec d'autres types entre `<>`

```
4 Paire<String , Double> p2 = new Paire<String , Double >("toto" , 3.0);  
5 String s2=p2.getEl1();  
6 Double y =p2.getEl2();
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

23/52

PROBLÉMATIQUE GÉNÉRALE (2/2)

Limites de la solution précédente :

- Obligation de faire des `cast` à chaque récupération d'objet
- Sécurisation **bof** : on peut mettre n'importe quoi dans la structure... + cast à la récupération
- Difficilement compatible avec des **algorithmes génériques** (tri, min, max...)

Solution (depuis Java 1.5) : utiliser des **génériques**

Classes, interfaces ou méthodes paramétrées par un ou plusieurs types, appelés **paramètres de type** (*type parameters*)

Exemple : type générique `ArrayList<E>` avec paramètre de type `E`

```
ArrayList<Integer> listeEntiers = new ArrayList<Integer >();  
ArrayList<Double> listeReels = new ArrayList<Double>();  
ArrayList<String> listeString = new ArrayList<String >();  
ArrayList<Point> listePoints = new ArrayList<Point >();
```

- Pas besoin de caster : `Point p = listePoints.get(0);` OK
- Seul les points sont acceptés dans la liste
- Une seule classe `ArrayList` ⇒ plus facilement compatibles avec différent algorithmes

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

20/52

CRÉATION D'UNE CLASSE GÉNÉRIQUE...

Exemple (très) utile : la **Paire**

La plupart des langages modernes gèrent des N-uplets... Mais pas Java. On peut créer une classe `Paire` pour retourner facilement plusieurs valeurs depuis une méthode.

```
1 public class Paire<A,B> { // 2 paramètres de type : A et B  
2     private A el1;  
3     private B el2;  
4     public Paire(A el1 , B el2) {  
5         this.el1 = el1;  
6         this.el2 = el2;  
7     }  
8     public A getEl1() {  
9         return el1;  
10    }  
11    public B getEl2() {  
12        return el2;  
13    }  
14 }
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

22/52

EXTENSION D'UNE CLASSE GÉNÉRIQUE (1)

Besoin d'une liste avec des méthodes spécifiques... Mais toujours générique

Exemple : récupération de la case du milieu

```
1 public class MaListeMilieu<E> extends ArrayList<E> {  
2     // constructeur sans argument par défaut  
3     // Les méthodes sont héritées : add , get , size ...  
4     // Méthode spécifique  
5     public E getMilieu(){  
6         return super.get(super.size()/2); // division entière  
7     }  
8 }  
9 }
```

Usage client :

```
1 MaListeMilieu<Double> li = new MaListeMilieu<Double >();  
2 li.add(2.0);  
3 li.add(1.4);  
4 li.add(3.7);  
5 System.out.println(li.getMilieu());
```

©2021-2022 C. Marsala / V. Guigue LU2IN002 - POO en Java

24/52

EXTENSION D'UNE CLASSE GÉNÉRIQUE (2)

Besoin d'une liste de *quelque chose*

- Aquarium = liste de poissons
- Train = liste de wagons
- Population = liste de personnes
- ...

```
1 public class Aquarium extends ArrayList<Poisson>{  
2     // bcp de méthodes héritées !!!  
3     ...  
4 }
```

Usage client : la liste ne gère que des poissons

```
1 Aquarium aqua = new Aquarium();  
2 aqua.add(new Thon());  
3 aqua.add(new Requin());
```

CAST SUR LES OBJETS GÉNÉRIQUES <E>

1 Coté *contenu* : très agréable (et classique)

- ◆ objets de types E et descendants de E
- ◆ récupération d'objets dans des variables E

2 Coté *contenant* : non flexible

```
ArrayList<Personne> pop = new ArrayList<Personne>(); // OK  
// Etudiant extends Personne  
ArrayList<Personne> promo = new ArrayList<Etudiant>(); // KO
```

⇒ Une seule issue (en cas de besoin) : la syntaxe *wildcard*

LA SYNTAXE *wildcard*

Subsommation *contenu/contenant*

On a besoin de cette propriété pour définir des algorithmes génériques. Syntaxe :

- `ArrayList<?>`
- ou `ArrayList<? extends Poisson>`

N'importe quelle liste, ou n'importe quelle liste d'objets dérivés de *poissons* :

```
ArrayList<? extends Poisson> li = new ArrayList<Thon>();
```

Exemple : comment proposer une technique de recherche de **minimum** dans une liste sans connaître le type de contenu ?

- 1 Définir une propriété (interface) : `Comparable`
- 2 Définir un algorithme acceptant n'importe quel conteneur d'objets comparables en utilisant la syntaxe *wildcard*

ATTENTION : ce type de syntaxe empêche toute modification sur l'objet passé

LA SYNTAXE *wildcard*

2 Utiliser la propriété dans un algorithme générique :

```
1 public class GenericTools<E extends Comparable<E>> {  
2     ...  
3  
4     public E getMinimum(ArrayList<? extends E> liste){  
5         E min = liste.get(0);  
6         for (int i=1; i<liste.size(); i++){  
7             // si : min > liste.get(i)  
8             if(min.compareTo(liste.get(i)) == 1)  
9                 min = liste.get(i);  
10        }  
11    }  
12    return min;  
13 }
```

Usage coté client :

```
1 ArrayList<Poisson> aquarium = new ArrayList<Poisson>();  
2 GenericTools<Poisson> tool = new GenericTools<Poisson>();  
3 Poisson lePlusPetit = tool.getMinimum(aquarium);
```

LA SYNTAXE *wildcard* (PRÉLIMINAIRES)

1 Définir une propriété : `Comparable`

```
1 public interface Comparable<E> {  
2     // retourne -1 si ref < obj , 0 si égalité, 1 sinon  
3     public int compareTo(E obj);  
4 }
```

Avec par exemple un Poisson répondant à la spécification :

```
1 public class Poisson implements Comparable<Poisson>{  
2     private double taille;  
3  
4     public Poisson(double taille) {  
5         this.taille = taille;  
6     }  
7     public double getTaille() {  
8         return taille;  
9     }  
10    public int compareTo(Poisson obj) {  
11        if(taille<obj.taille)  
12            return -1;  
13        else if(taille==obj.taille)  
14            return 0;  
15        else  
16            return 1;  
17 }
```

ALGORITHMES GÉNÉRIQUES

Classe algorithmique de gestion des listes générique

Collections

- `min, max, sort, shuffle, indexOf, frequency...`

Quelques exemples d'outils disponibles :

static int frequency(Collection<? extends Object>, Object o)	Returns the number of elements in the specified collection equal to the specified object.
static int indexOf(List<? extends Object>, Object o)	Returns the starting position of the first occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)	Returns the minimum element of the given collection, according to the natural ordering of its elements.
static void shuffle(List<? extends Object> list)	Randomly permutes the specified list using a default source of randomness.
static <T extends Comparable<? super T>> void sort(List<? extends T> list)	Sorts the specified list into ascending order, according to the natural ordering of its elements.
static <T> void sort(List<? extends T> list, Comparator<? super T> c)	Sorts the specified list according to the order induced by the specified comparator.

- 1 Flux (suite)
- 2 Conteneurs génériques
- 3 Design Patterns
- 4 Exercices d'annales : Examen Janvier 2020 (2019oct)

Principes Objets :

- Encapsulation
- Héritage & abstraction
- Polymorphisme

Question (difficile)

Comment combiner ces différents outils pour écrire des programmes ou boîtes-à-outils flexibles, réutilisables, efficaces, etc... ?

PATRONS DE CONCEPTION (DESIGN PATTERNS)**Définition : design pattern**

Élément de conception réutilisable permettant de résoudre un problème récurrent en programmation orientée objet.

■ Historique :

- ◆ analogie avec une méthode de conception d'immeuble en architecture [Alexander, 77]
- ◆ introduites par le "GOF" dans le livre Design Patterns en 1999
 - dans le "GOF" 23 patterns standards

Pourquoi les patterns ?

Des **recettes d'expert** ayant fait leurs preuves.

Un vocabulaire commun pour les architectes logiciels.

Approche incontournable dans le monde de la P.O.O.

Remarque : certains patterns sont déjà inclus dans le langage...

IDIOMES DE LA PROGRAMMATION

Portion de code Java que l'on utilise lorsque l'on a à résoudre un problème récurrent

■ Parcours d'un tableau

```
1 for (int i=0;i2     tableau[i] = ...  
3 }
```

■ Gestion d'exception

```
1 try {  
2     XYZ(...);  
3 } catch(XYZException e) {  
4     e.printStackTrace(System.err);  
5 }
```

CONCEPTION VS. PROGRAMMATION

Un design pattern est un élément de **conception** (objet)

Remarque : ce n'est pas au niveau du langage de programmation (donc pas spécifique à Java, un pattern est aussi valable en C++, en Python...)

Citations

"Chaque patron décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le cœur de la solution à ce problème, d'une façon telle que l'on puisse réutiliser cette solution des millions de fois, sans jamais le faire deux fois de la même manière" Christopher Alexander - 1977.

"Les patrons offrent la possibilité de capitaliser un savoir précieux né du savoir-faire d'experts" [Buschmann] - 1996.

COTÉ IMPLÉMENTATION : GRANDS PRINCIPES**Principe 1**

Favoriser la **composition** (lien dynamique, flexible) sur **l'héritage** (lien statique, peu flexible) La délégation est un exemple d'outil pour la composition

Attention : favoriser ne veut pas dire remplacer systématiquement, l'héritage est largement utilisé aussi !

Principe 2

Les clients programment en priorité pour des **interfaces** (ou **abstractions, classes abstraites, etc**) plutôt qu'en lien direct avec les implémentations (classes concrètes)

DIFFÉRENTS TYPES DE PATTERNS

- **Pattern de création** : utilisé pour la création d'objets
 - ◆ par exemple : singleton, prototype, factory,...
- **Pattern structurel** : utilisé pour rajouter des fonctionnalités à une classe
 - ◆ par exemple : decorator, composite,...
- **Pattern comportemental** : utilisé pour mettre en œuvre des moyens de communication entre objets
 - ◆ par exemple : iterator, strategy,...

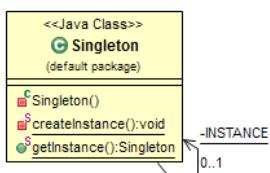
DESCRIPTION STANDARD

Nom (classification)	singleton (création)
Intention	description générale et succincte
Alias	autres noms connus pour le pattern
Motivation	au moins 2 exemples/scénarios
Indications d'utilisation	liste des situations qui justifient de l'utilisation du pattern
Structure	diagramme de classe UML indépendant du langage
Constituants	explication des différentes classes intervenant dans le pattern
Implémentation	principes, pièges, astuces, techniques pour planter le pattern dans un langage objet donné
Utilisations remarquables	programmes réels qui l'utilisent
Limites	limites concernant son utilisation

SINGLETON (CONSTRUCTION)

Garantir l'unicité d'une instance

- A utiliser quand il ne peut y avoir qu'une instance (eg Console)
- Fournir un accès à cette instance



SINGLETON (CONSTRUCTION)

```

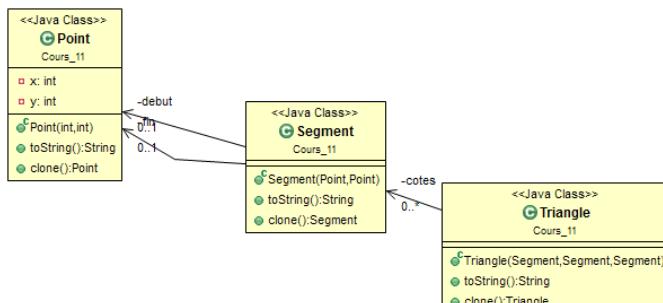
1 public class Singleton {
2     private static Singleton INSTANCE = null;
3
4     private Singleton() {}
5
6     private static void createInstance() {
7         if (INSTANCE == null) {
8             INSTANCE = new Singleton();
9         }
10    }
11    public static Singleton getInstance() {
12        if (INSTANCE == null) createInstance();
13        return INSTANCE;
14    }
15 }
  
```

Dans cette version de Singleton, l'instance est créée seulement si la méthode `getInstance()` est appelée au moins une fois

PROTOTYPE (CONSTRUCTION)

Créer un nouvel objet à partir d'un objet existant

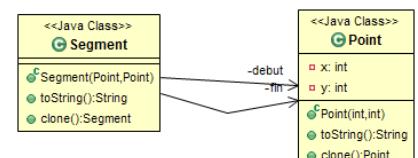
- Fournir une **copie** exacte de l'objet existant
- Fournir une interface commune aux objets



COMPOSITE (STRUCTURE)

Un objet E, composé d'objets E

- une sous-figure, composée de figure
- une stratégie, composée de sous-stratégie



```

1 public class Segment {
2     private Point debut, fin;
3     public Segment(Point p1, Point p2) {
4         debut = p1; fin = p2;
5     }
6
7     public String toString() {
8         return "Segment [debut=" + debut + ", fin=" + fin
9     }
10    public Segment clone() {
11        return new Segment(debut.clone(), fin.clone());
12    }
13 }
  
```

ITERATOR (COMPORTEMENT)

Objectif

- Découpler le choix des structures de données des implémentations d'algorithmes
- Proposer une interface de parcours d'un ensemble d'objets quelle que soit la structure associée (liste, pile, arbre,...)

```

1 // code d'utilisation spécifique
2 Iterator<MyType> iter = list.iterator();
3
4 // code commun à toute structure de données
5 while (iter.hasNext()) {
6     System.out.print(iter.next());
7     if (iter.hasNext())
8         System.out.print(", ");
9 }

```

DECORATOR (STRUCTURE)

Ajouter une fonctionnalité...

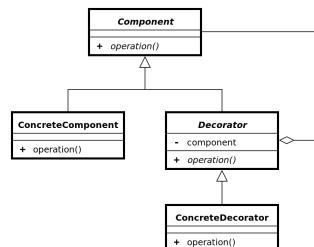
... sur n'importe quel objet d'une arborescence de classe

- Permettre de lire des informations de haut niveau (String, double...) dans n'importe quel flux
- Rendre n'importe quelle stratégie prudente

```

1 public MonObjetDecor extends MonObjet{
2     private MonObjet obj;
3     public MonObjetDecor(MonObjet obj){ this.obj = obj;}
4
5     public void mafonction(){
6         if(cas1) return obj.mafonction()
7         else // code spécifique
8     }

```



ET PLEIN D'AUTRES ENCORE

- Visitor
 - vient exécuter un algorithme dans un objet
- MVC : model view controller
 - pour les interfaces graphiques, séparation des éléments clés
- ...

Comme un second niveau de programmation (de la conception en fait !)

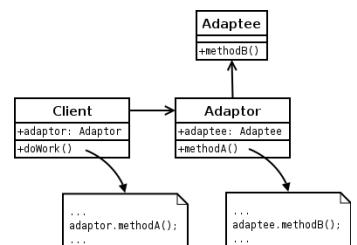
A continuer avec de l'UML et d'autres UE de génie logiciel

- UE L3 LU3IN002 "Programmation par objets"

ADAPTER (STRUCTUREL)

Idée : réutiliser une fonction déjà implémentée...

... Mais dans une architecture contrainte
= Adapter la classe
Repose sur le principe de la délégation

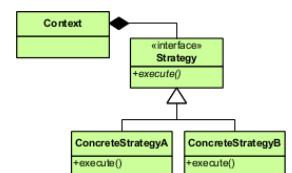


- Très très fréquent : réutiliser une classe sans la modifier...
Alors que le client est déjà spécifié
 - ♦ on a un objet d'une classe Client qui veut utiliser un objet de classe B mais le client ne sait manipuler des objets de classe A.
Il faut donc adapter la classe B à l'interface de A pour que le Client puisse finalement utiliser B.
- Technique d'optimisation, calcul de graphes...
 - ♦ Adapter les objets à passer en paramètres... Ou adapter les algorithmes.

STRATEGY (COMPORTEMENT)

Gérer le comportement distinctement de l'objet

- Robot... Qui marche, rampe, vole...
Ou une combinaison des 3
- Le client gère la stratégie
- On peut créer de nouvelles stratégies avec des robots existants



```

1 public class Robot{
2     private Strategy str;
3     public Robot(Strategy str){this.str = str;}
4
5     public void action(){
6         str.execute(); // ou str.execute(this);
7     }

```

⇒ Une alternative (beaucoup plus flexible) à l'héritage sur les robots

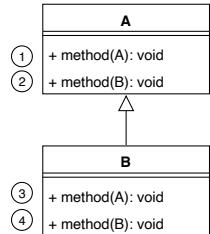
Note : DP compatible avec Composite

PROGRAMME

- 1 Flux (suite)
- 2 Conteneurs génériques
- 3 Design Patterns
- 4 Exercices d'annales : Examen Janvier 2020 (2019oct)
 - Exercice 1 : Combien d'instances, quelle méthode?

EXAMEN JANVIER 2020 (2019OCT) : EXERCICE 1

Exercice 1 : Combien d'instances, quelle méthode?



Question 1

```

1 A[] tab = new A[5];
2 for(int i=0; i<4; i++) {
3     if (i==0)
4         tab[i] = new A();
5     else
6         tab[i] = new B();
7 }
8 tab[1] = tab[0];
9 A a = new A();
10 tab[2] = a;
  
```

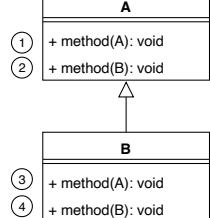
- Combien d'instances de A et de B ont été créées au total?
Donner les 2 chiffres.

- Combien en reste-t-il à l'issue de l'exécution du programme?

Solutions

- 2 instances de A (ligne 4 (pour i=0) et ligne 9)
3 instances de B (3 fois pour i=1,2,3)
- Les instances référencées par tab[1] et tab[2] ont été déréférencées (lignes 8 et 10), il reste donc 2 instances de A et 1 instance de B

EXAMEN JANVIER 2020 (2019OCT) : EXERCICE 1



Question 3 : dans le cas où la méthode ② n'existe plus

```

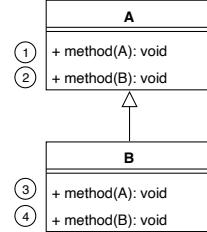
1 A a=new A();
2 B b=new B();
3 A ab=new B();
4
5 ab.method(a);
6 ab.method(b);
7 ab.method(ab);
  
```

- Dans les lignes 5, 6 et 7, quelles sont les méthodes pré-sélectionnées par le compilateur?
- Quelles sont les méthodes exécutées par la JVM?

Solutions

- Présélection : ① ① ① (une seule méthode disponible à partir de la variable ab de type A)
- Exécution : ③ ③ ③ (dépend de l'instance)

EXAMEN JANVIER 2020 (2019OCT) : EXERCICE 1



Question 2

```

1 A a=new A();
2 B b=new B();
3 A ab=new B();
4
5 ab.method(a);
6 ab.method(b);
7 ab.method(ab);
  
```

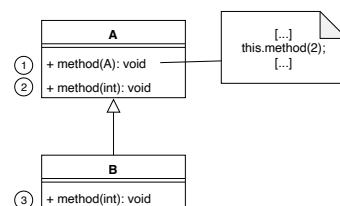
- Dans les lignes 5, 6 et 7, quelles sont les méthodes pré-sélectionnées par le compilateur?

- Quelles sont les méthodes exécutées par la JVM?

Solutions

- Présélection : ① ② ① (la variable ab est de type A, elle n'a accès que aux méthodes de A)
- Exécution : ③ ④ ③ (dépend de l'instance référencée par ab qui est de type B)

EXAMEN JANVIER 2020 (2019OCT) : EXERCICE 1



Question 4

```

1 A a=new A();
2 B b=new B();
3 A ab=new B();
4
5 a.method(a);
6 b.method(a);
7 ab.method(a);
  
```

- En considérant la nouvelle architecture. Quelle méthode est invoquée à l'intérieur de ① dans les 3 appels?

Solutions

- Exécution : ② ③ ③ (dépend de l'instance)