Conception OO Avancée master pro

Pr. Laid KAHLOUL

2023

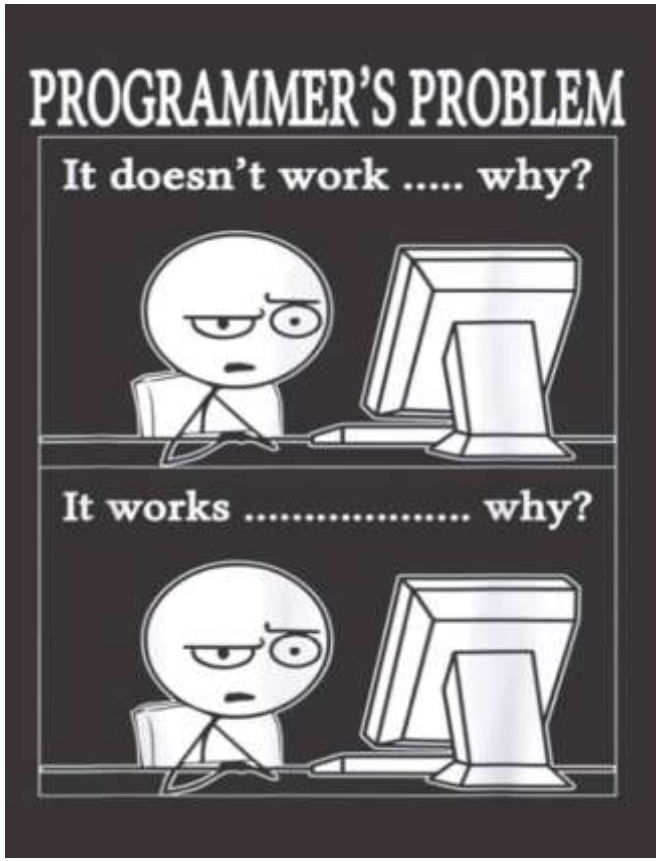
Contenu

- Rappel +
- Rappel sur Héritage
- Structure de données et **généricité**
- Héritage et polymorphisme
- **Exceptions**
- Principe **SOLID** (**S**ingle responsability principle, **O**pen/closed principle, **L**iskov substitution principle, **I**nterface segregation principle, **D**ependency inversion principle)
- **Design Patterns**

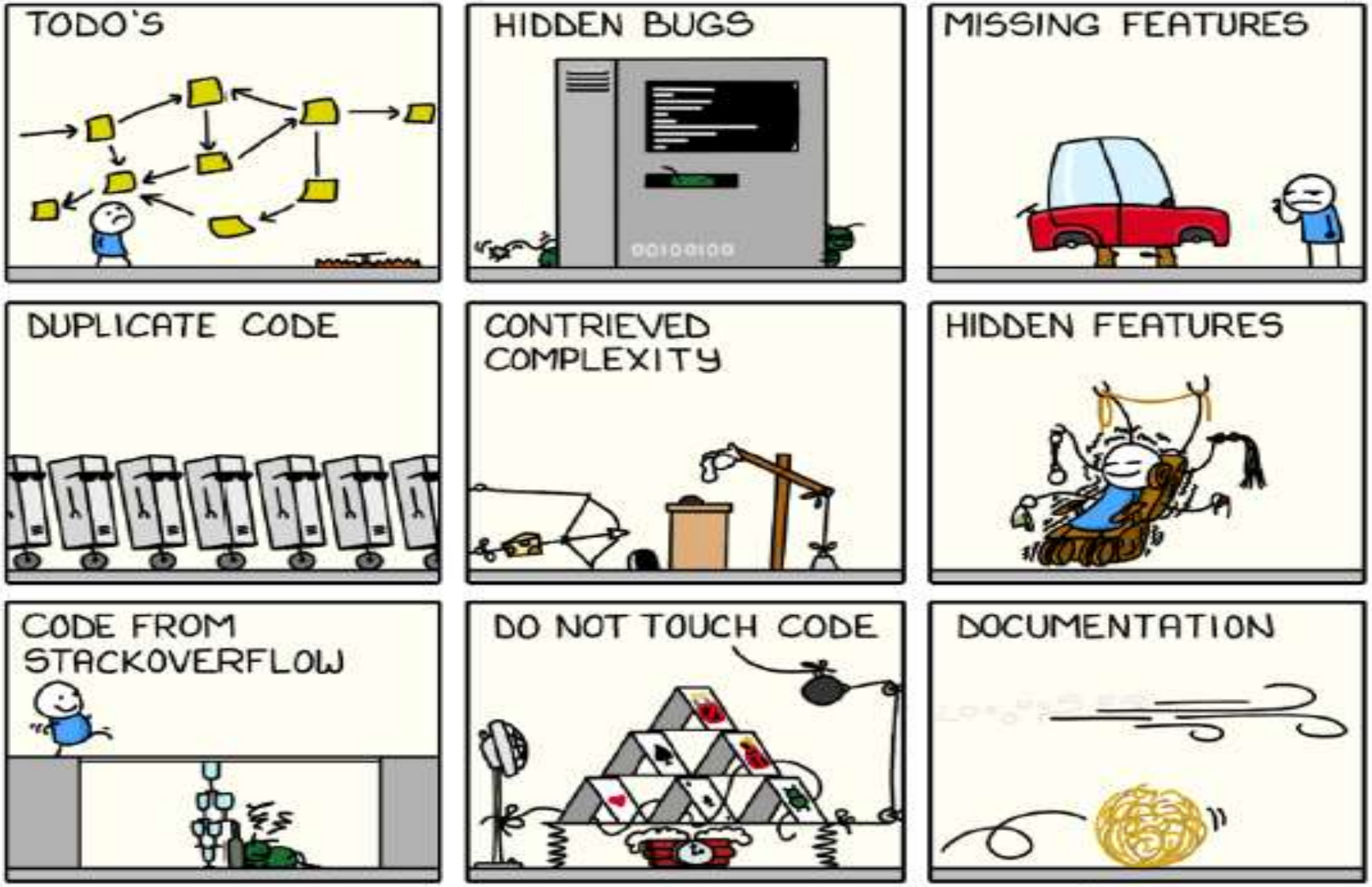
Conception OO Avancée
Partie1:

Rappel et +
https://www.lirmm.fr/~pvalicov/Teaching.html

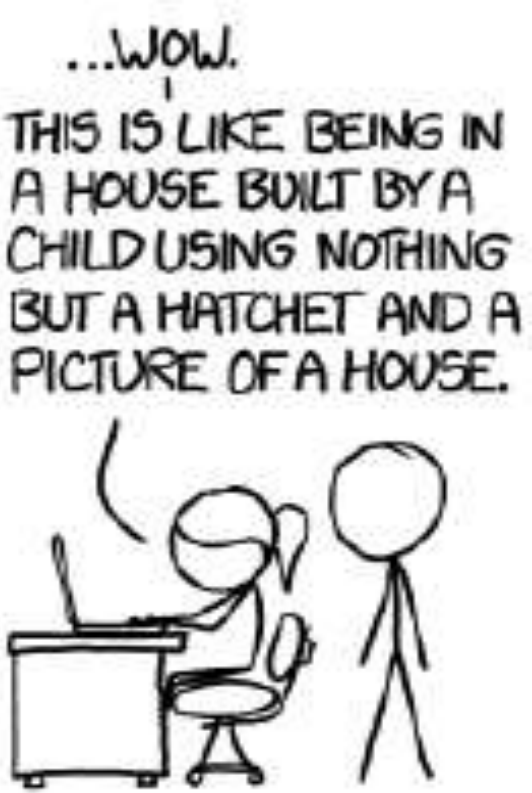
Before starting

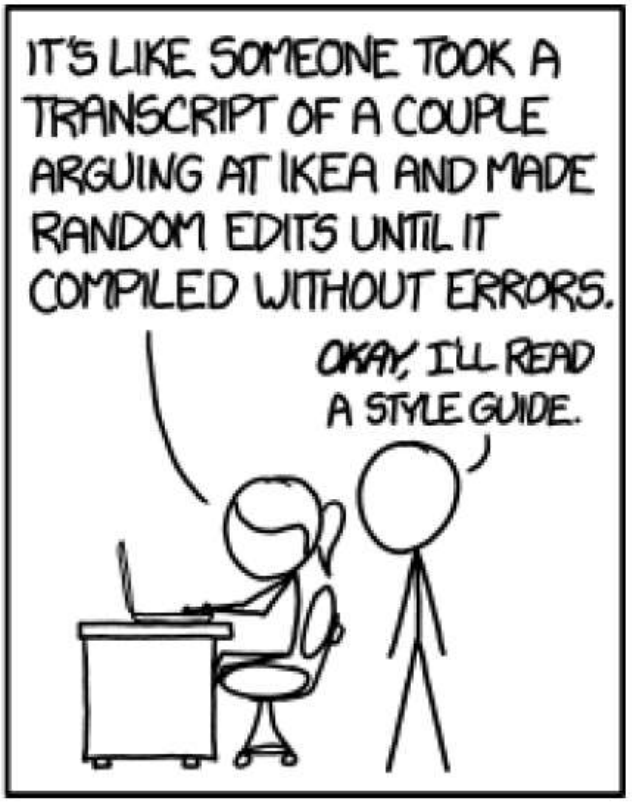


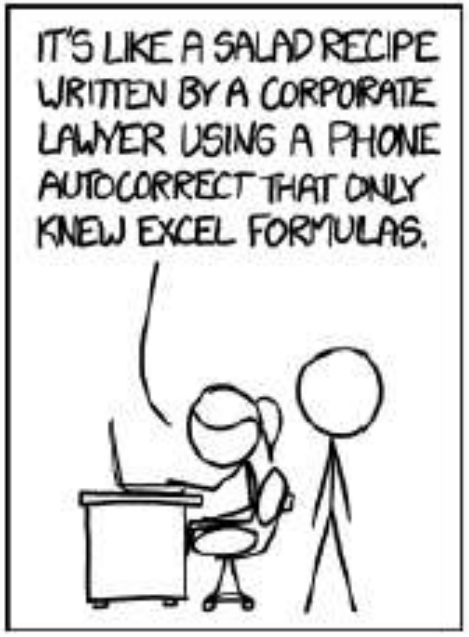
POSSIBLE CODE CONTENTS



KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.
LEMME SEE- I'M SURE IT'S FINE.


...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.


IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.
OKAY, I'LL READ A STYLE GUIDE.


IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.

Contenu de ce chapitre

1. **Java** et **UML**: c'est quoi ? un rappel, compilation, standards, diagrammes.
2. **Approches de programmations** : fonctionnele, OO, objet, classe, construction, constructeur par défaut, destruction,
3. **Types en Java** : primitif vs non primitif, classes enveloppe, utilité des primitifs ?, accès par valeur ou par référence, passage de paramètre, reference this
4. Membres **static** & **final**
5. Environnement java : **package**, **encapsulation** et **visibilité**
6. **Relations entre classe** : dépendance, association, cardinalité des associations, navigabilité, agrégation et composition
7. Types **énuméré** et **record**
8. **Règles pour écrire du bon code** : règles générales, convention de nommage (sémantique et lexique), PRU (principe de la responsabilité unique)

Objectifs

- Approfondir les notions de conception/programmation objet
- Mise en œuvre des acquis à travers un langage orienté objets (`Java`)
- Apprendre à programmer "proprement"
- Écriture des tests unitaires (avec `JUnit`)
- Utilisation d'un gestionnaire des versions (avec `Git`)

} **plutôt en TP**

- Savoir interpréter et concevoir des diagrammes de classes

Java et UML: C’est quoi?

Rappel sur Java: c'est quoi?

Java

Java est un langage de programmation

- langage de haut niveau, objet, portable, ...
- langage pseudocompilé (*bytecode*) exécuté par la *JVM*

Java est aussi une plateforme

- plateforme portable (*Write once, run anywhere*)
- interface de programmation d'application (API)
  - Swing, AWT, JavaFX
  - JDBC
  - Réseau, HTTP, FTP
  - IO, math, config, etc.

Pourquoi Java dans ce cours ?

- Simple à prendre en main
  - l'API officielle très riche et assez fiable,
  - langage compilé et robuste :
    langage fortement typé
    avec un mécanisme puissant de gestion d'exceptions
  - pas très verbeux
  - gestion automatique de la mémoire :
    utilisation implicite des pointeurs (références)
    le ramasse-miettes (*garbage collector*)
    } plus de sécurité
- framework de tests unitaires complet : `JUnit`
- un des langages orientées objets les plus répandus au monde

Attention : ce cours aurait pu très bien avoir un autre langage orientée objet comme support (`C++`, `C#`, `Python`, etc.)

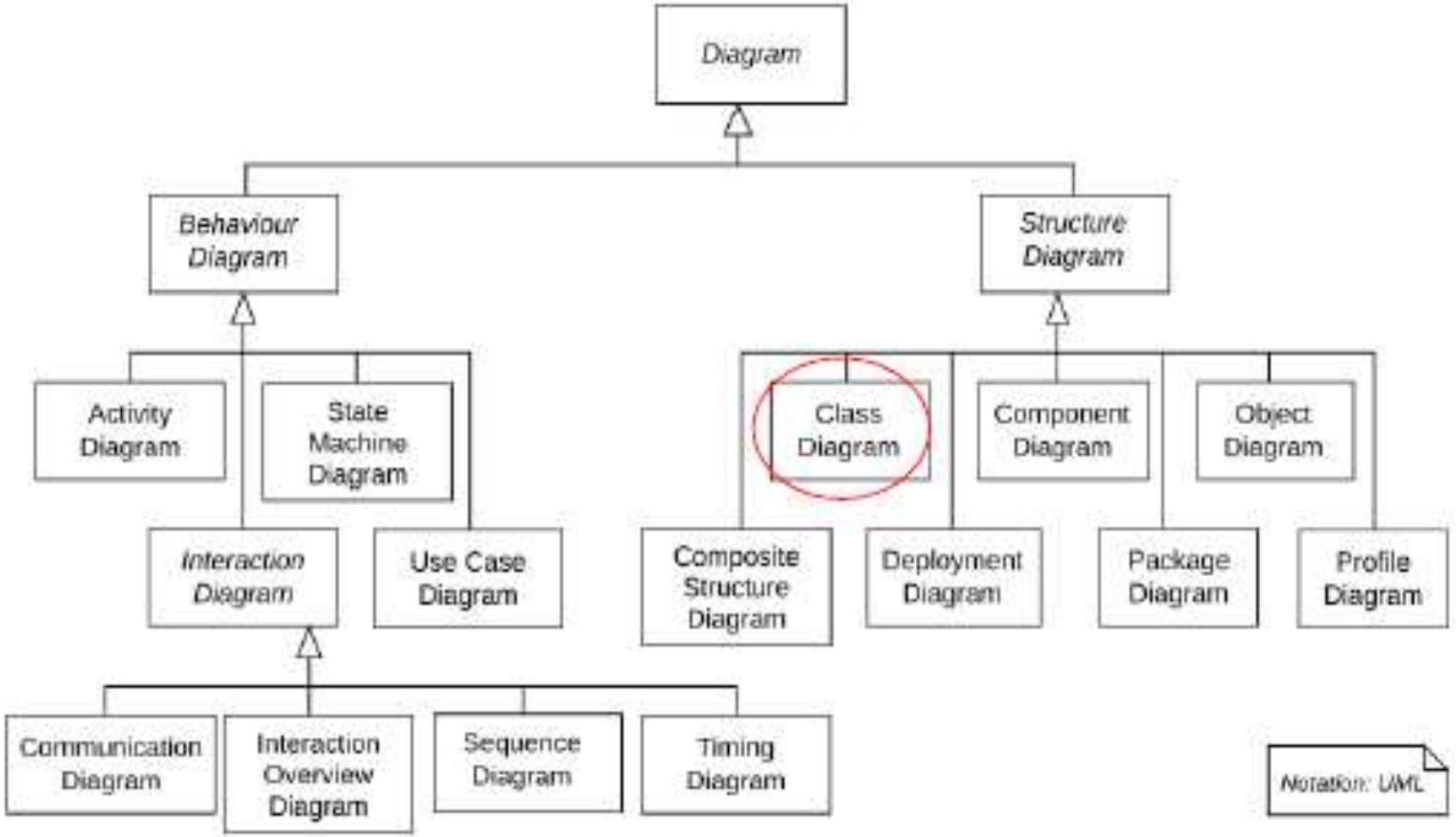
UML - Unified Modeling Language

Définition

UML est un **langage de modélisation** orienté objet qui permet de représenter (de manière graphique) et de communiquer les divers aspects d'un système informatique.

- Apparu au milieu des années '90
- Version actuelle : UML 2.x (standard ISO adopté par l'OMG)
- **langage de modélisation** $\neq$ **langage de programmation**
- C'est juste un ensemble de notations ayant comme base la notion d'*objet*

Les diagrammes UML



Approches de programmations

Approches de programmation

Approche **fonctionnelle**

- Approche traditionnelle utilisant des procédures et des fonctions
- On identifie les fonctions nécessaires à l'obtention du résultat
- Les grands programmes sont décomposés en sous programmes
  - hiérarchie de fonctions et sous-fonctions
  - approche descendante (Top-Down)

Approche **orientée objet**

1. On identifie les *entités* (objets) du système
2. Chaque objet possède un ensemble de compétences (fonctions)
3. On cherche à faire *collaborer* ces objets pour qu'ils accomplissent la tâche voulue :
   - les objets interagissent en s'envoyant des messages (appels de fonctions)
   - un message dit quoi faire, mais non comment : chaque objet est responsable de son fonctionnement interne

Objets - première immersion

Définition

Un **objet** informatique définit une représentation simplifiée, une abstraction d'une entité du monde réel.

But recherché :

- mettre en avant les caractéristiques essentielles
- dissimuler les détails

Exemple d'objet *Etudiant* (ou abstraction *Etudiant*)

- description : *nr. d'étudiant, année d'inscription, notes en cours, absences...*
- compétences (ou services) : *apprendre, s'inscrire, passer les examens...*

Objets - première immersion

Définition (une autre)

Objet = identité + état + comportement

Exemple d'objet *Voiture* :

- Identité : numéro d'identification ou code-barres etc.
- État : marque, modèle, couleur, vitesse...
- Services rendus par l'objet : *Démarrer*, *Arrêter*, *Accélérer*, *Freiner*, *Climatiser*, ...

Lorsqu'on utilise objet, on n'a pas à connaître/comprendre son fonctionnement interne :

- à priori on ne connaît pas les composantes d'une voiture...
- à priori on ne connaît pas les mécanismes internes de la voiture (comment elle accélère, la façon dont elle freine, etc)...
- ...mais on peut *utiliser* la voiture en lui *demandant* ses services !

Classe *vs* Objets

Définition

Une classe est une description abstraite d'un ensemble d'objets "*de même nature*". Ces objets sont des **instances** de cette classe.

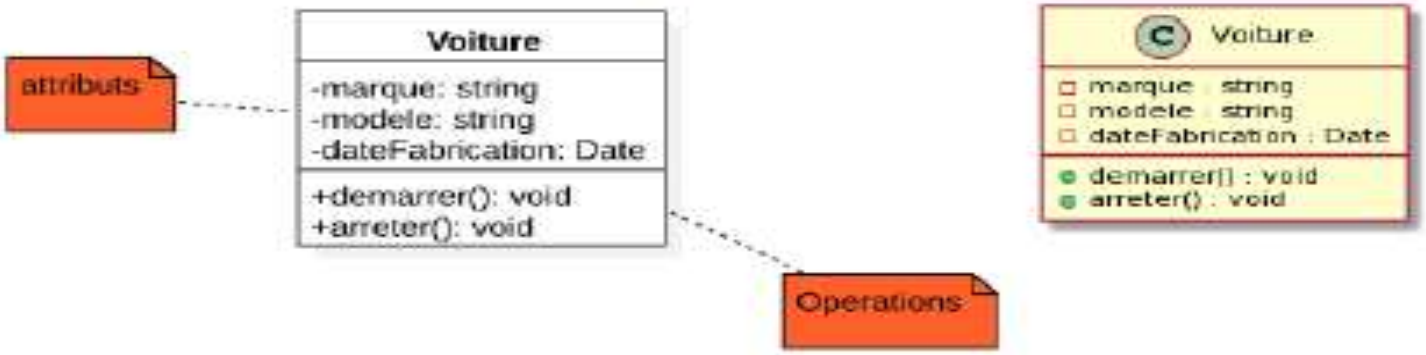
- La classe est un "**modèle**" pour la création de nouveaux objets.
- "**Contrat**" garantissant les compétences de ses objets.

Exemples :

- L'objet *philippe* est une instance de la classe `Humain`.
- L'objet *renaultQuatreL* est une instance de la classe `Voiture`.
- L'objet *bases_Prog_Objet* est une instance de la classe `Cours`.

Classe *vs* Objet – exemples

Illustration d'une classe graphiquement (en UML) :

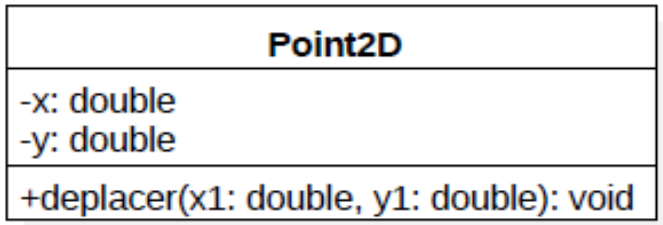


La même classe écrite en Java :

```java
class Voiture {
    private String marque;
    private String modele;
    private LocalDate dateFabrication;

    public void demarrer() {
       /* Le code à écrire au moment où on programme */
    }

    public void arreter() {
       /* Le code à écrire au moment où on programme */
    }

}
```

Classe vs Objet – parallèle avec les BD

En UML :



En Java :

```java
class Point2D {
    private double x;
    private double y;

    public void deplacer(double x1, double y1) {
    /* Le code des méthodes à écrire
    au moment où on programme */
    }

}
```

En base de données :

ID	x	y
0	1.2	-0.6
1	0.5	0.5
2	0	1
3	3.4	0.3

- Table $\approx$ Classe
- Chaque ligne est un objet (instance de la classe)
- Important : **pas d'équivalent des méthodes**

Construction des objets

Dans un programme Java, toute variable doit être **initialisée** avant d'être utilisée.

Pour les objets on parle de **construction**.

```
class Voiture {
    private String marque;
    private String modele;

    public void demarrer() {
       /* Le code de la fonction */
    }

    public void arreter() {
       /* Le code de la fonction */
    }

}
```

```
class Voiture {
    private String marque;
    private String modele;

    // constructeur explicite
    public Voiture(String uneMarque, String unModele) {
       marque = uneMarque;
       modele = unModele;
    }

    public void demarrer() {
       /* Le code de la fonction */
    }

    public void arreter() {
       /* Le code de la fonction */
    }

}
```

```
// construction par défaut
Voiture renaultQuatreL = new Voiture();
```

```
// construction explicite
Voiture renaultQuatreL = new Voiture("Renault", "QuatreL");
```

Construction des objets

Le constructeur est une fonction Java très particulière :

- pas de type de retour
- le nom est identique au nom de la classe correspondante
- sont invoqués avec l'opérateur `new`

```
Voiture vehicule = new Voiture("Renault", "QuatreL");
```

Principe de fonctionnement :

1. l'opérateur `new` crée une nouvelle instance de la classe
   - alloue d'un bloc mémoire pour le futur objet
   - retourne la référence vers cet objet

   $\Longrightarrow$ naissance de l'instance `this`
2. la méthode de construction est appelée sur la nouvelle instance

Le corps du constructeur sert à initialiser les valeurs de chaque attribut de l'objet `this`

Construction par défaut

Si le développeur n'a écrit **aucun constructeur**, alors le compilateur fournit un constructeur sans argument qui consiste à initialiser tous les champs avec leur valeur par défaut.

```
class Point2D {
    int x, y;

    public String toString() {
        return "[" + x + "," + y + "]";
    }
}
```

=

```
class Point2D {
    int x, y;

    public Point2D() {
        x = 0;
        y = 0;
    }

    public String toString() {
        return "[" + x + "," + y + "]";
    }
}
```

À l'utilisation :

```
Point2D point = new Point2D();
System.out.println(point); // [0,0] s'affiche
```

L'écriture d'un constructeur explicite suspend le mécanisme de construction par défaut.

Collaboration entre constructeurs

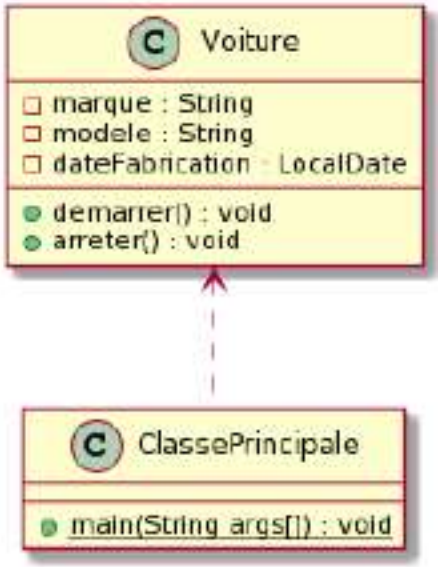
Une classe peut avoir plusieurs constructeurs :

```
class Compte {
    private String nom, prenom;
    private double soldeCompte;

    public Compte(String n) {
       nom = n;
    }

    public Compte(String n, String p) {
       this(n); // appel du constructeur Compte(String)
       prenom = p;
    }

    public Compte(String n, String p, double solde) {
       this(n, p); // appel du constructeur Compte(String, String)
       soldeCompte = solde;
    }

    public void rechargerCompte(double solde) {
       soldeCompte = solde;
    }
}
```

Exemple d'une application orientée objet en Java

```java
class Voiture {
    private String marque;
    private String modele;
    private LocalDate dateFabrication;

    // le constructeur de la classe
    public Voiture(String uneMarque, String unModele) {
       marque = uneMarque;
       modele = unModele;
       dateFabrication = LocalDate.now();
    }

    public void demarrer() {
       /* Du code */
    }

    public void arreter() {
       /* Du code */
    }
}
```

Illustration en UML :



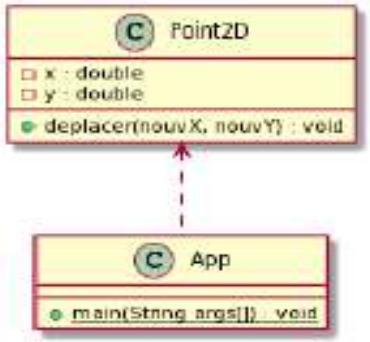

```java
class ClassePrincipale {

    /* Méthode principale de l'application Java : le point d'entrée du programme */
    public static void main(String args[]) {
       Voiture maVoiture = new Voiture("Renault", "QuatreL"); // Création d'un objet maVoiture
       maVoiture.demarrer(); // Utilisation d'un service de l'objet
       maVoiture.arreter(); // Utilisation d'un service de l'objet
    }

}
```

Un autre exemple

```java
class Point2D {

    private double x;
    private double y;

    public Point2D(){
       x = 1;
       y = 2;
    }

    public void deplacer(double nouvX, double nouvY) {
       x = nouvX;
       y = nouvY;
    }

}
```

```java
class Point2D {

    private double x;
    private double y;

    public Point2D(){
       x = 1;
       y = 2;
    }

    public void deplacer(double nouvX, double nouvY) {
       x = Math.abs(nouvX);
       y = nouvY;
    }

}
```

```java
class App {
    public static void main(String args[]) {

        // Création d'un point
        Point2D petitPoint = new Point2D();

        // Utilisation du point créé
        petitPoint.deplacer(2,3);

        petitPoint.deplacer(-6,4);
    }
}
```



: Illustration en UML

L'utilisation de l'objet `petitPoint` se fait de la même manière quelques soient les changements du code de son fonctionnement interne.

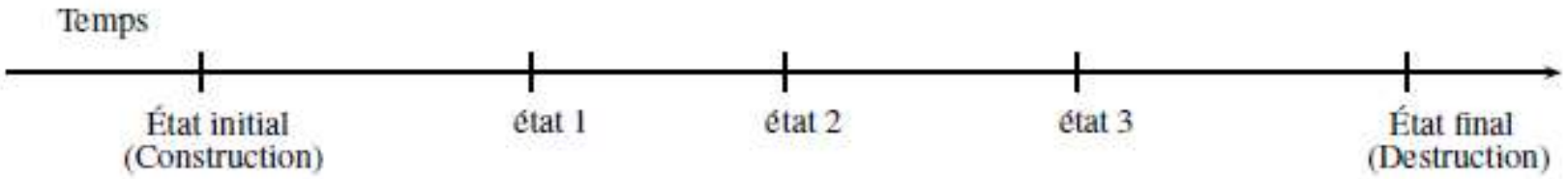
Petite recap – vocabulaire

Pour une classe :

- **attribut** = champs $\approx$ variable globale
- **fonction** = procédure = **méthode**
- **construction** = **instanciation** $\approx$ initialisation "propre"

Lorsqu'un objet *o* d'une classe *A* est construit, on dit que *o* est une instance de la classe *A*.

Cycle de vie des objets



Dans une application, le cycle de vie d'un objet a 3 phases :

1. Création de l'objet en faisant appel au **constructeur**
   - Réservation d'un bloc mémoire correspondant à l'objet
   - Initialisation des ses attributs : état initial
2. L'**évolution** de l'objet où il peut changer d'états à plusieurs reprises
3. **Destruction** de l'objet (implicite en Java)
   - invoquée lorsque l'objet n'a plus vocation d'être utilisé
   - libération de l'espace mémoire utilisé par l'objet
   - se fait "automatiquement" dans certains langage (dont Java), il suffit "d'oublier" l'objet – *Garbage Collector* ou ramasse-miettes

Destruction des objets

- En Java, le Garbage Collector (ramasse-miettes) est responsable de la destruction des objets inutiles.

- Le Garbage Collector se déclenche en temps "presque masqué", lorsque la mémoire manque ou la machine virtuelle est oisive.

- Principe de fonctionnement :
  1. marquage des cellules mémoires occupées par des objets **accessibles**
  2. tout ce qui est non marqué est désigné "garbage"
  3. désignation des cellules mémoires non marquées comme disponibles à la réutilisation (on peut écraser leur contenu)

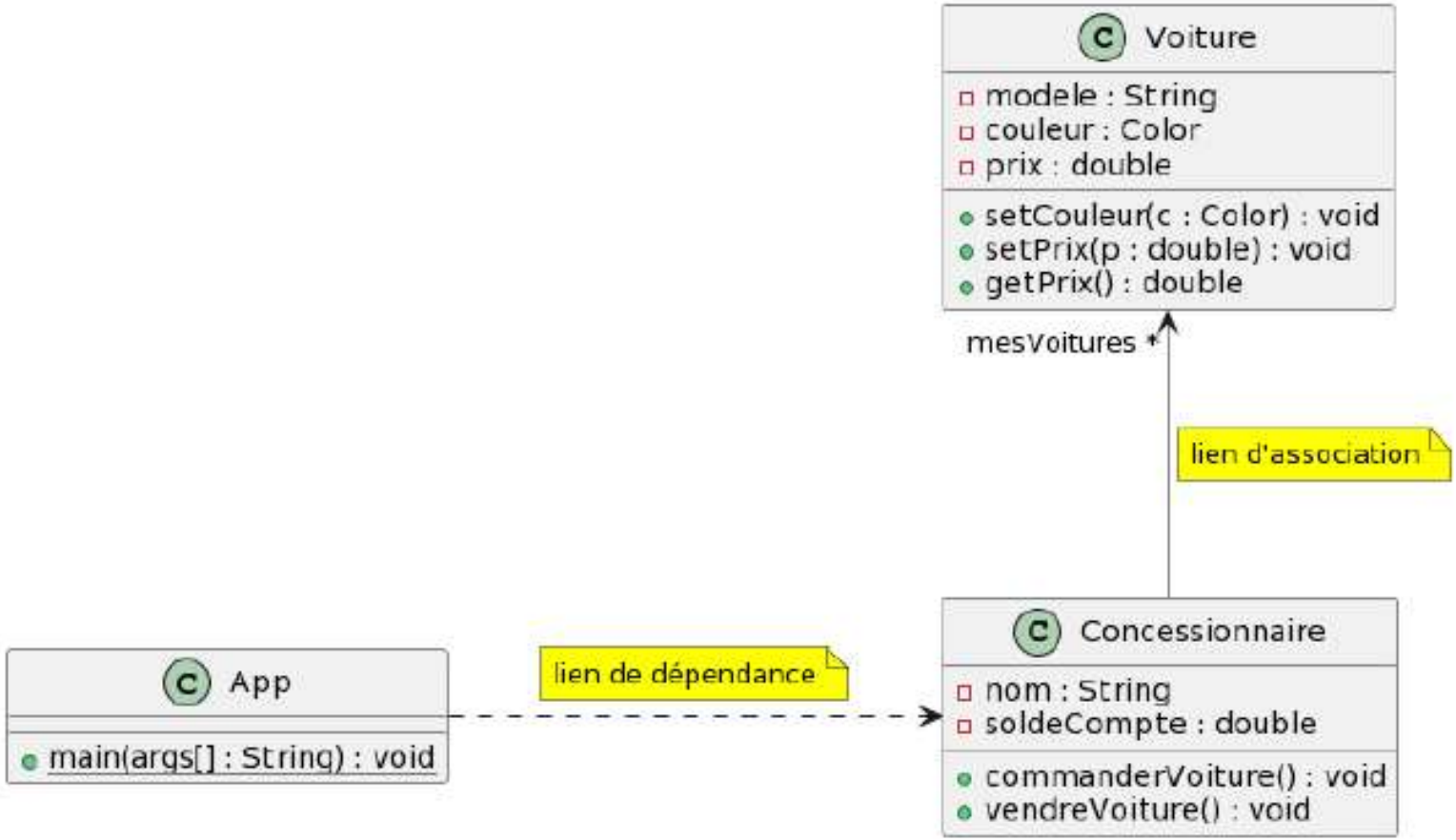
- **Important** : dès lors que l'objet n'est plus accessible dans le code, il faut supposer qu'il est détruit.

```java
import java.awt.Color;

public class Voiture {
    private String modele;
    private Color couleur;
    private double prix;

    public Voiture(String modele, double prix) {
        this.modele = modele;
        this.prix = prix;
        this.couleur = Color.WHITE;
    }

    public void setCouleur(Color couleur) {
        this.couleur = couleur;
    }

    public void setPrix(double prix) {
        this.prix = prix;
    }
```

```java
import java.util.ArrayList;
import java.awt.Color;

public class Concessionnaire {
    private String nom;
    private double soldeCompte;
    private ArrayList<Voiture> mesVoitures;

    public Concessionnaire(String nom, double solde) {
        this.nom = nom;
        soldeCompte = solde;
        mesVoitures = new ArrayList<>();
    }

    public void commanderVoiture(){
        Voiture v = new Voiture("SuperCitadine", 0);
        v.setCouleur(Color.RED); // changement d'état
        v.setPrix(10255); // changement d'état
        mesVoitures.add(v);
    }

    public void vendreVoiture(){
                                    itures.get(0);
                                    getPrix();
                                    de la liste la première
                                    la référence v
                                    e(v);
```

```java
    pu public class App {
          public static void main(String args[]) {
    }         Concessionnaire treNaud = new Concessionnaire("treNaud", 125360);
}
              // méthode où un objet de type Voiture sera créé
              treNaud.commanderVoiture();

              treNaud.vendreVoiture();

             /* À la fin de l'exécution de la méthode vendreVoiture(), un objet
              de type Voiture ne sera plus accessible. Donc, l'espace mémoire
              qu'il occupe sera rendu disponbile par le Garbage Collector. */
          }
      }
```

Illustration en UML de l'exemple précédent



Classe–Type–Objet

Définition

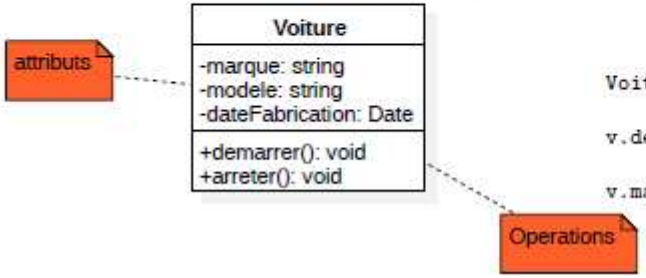
Un type de données définit :

- un domaine de valeurs (associé aux données de ce type)
- un ensemble d'opérations applicables aux données de ce type

```
int n = 40; // n est de type int
n = n + 2; // l'opération + est applicable sur les int
boolean b = true; // b est de type boolean
b = true/3: // opération non-valide car la division n'est pas applicable sur les boolean
```

Une classe définit en fait un **type** d'objets :

- un ensemble de propriétés (des attributs)
- un ensemble de comportements applicables (des opérations)




```
Voiture v; // v est de type Voiture

v.demarrer(); // méthode applicable sur un type Voiture

v.manger(); // méthode non-applicable sur un type Voiture
```

Typage Java

En Java toutes les données sont *typées*. Il y a deux sortes de types :

- types primitifs (prédéfinis)
- types objets (référence)

type primitif	taille en octets	valeur par défaut
boolean	1 bit	false
byte	1	0
short	2	0
int	4	0
long	8	0
float	4	0.0
double	8	0.0
char	2	'\u0000'

Accès **par valeur** pour les types primitifs :

```
int x, y;
x = 2;
y = 3;

// on compare la valeur de x à la valeur de y
x == y;
```

L'accès aux types objets se fait **par référence** :

```
Voiture x, y; // deux objets de type Voiture
x = new Voiture();
y = new Voiture();
x == y; // on compare les adresses mémoire de x et de y
```