Développement Orienté Objets
Introduction

Petru Valicov
petru.valicov@umontpellier.fr

https://gitlabinfo.iutmontp.univ-montp2.fr/dev-objets

2023-2024

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

Planning du cours - 1h par séance

1. Généralités sur les objets : Java, UML (≈ 4-5 cours)
2. Héritage et Polymorphisme (≈ 3 cours)
3. Structures de données et Généricité (≈ 4 cours)
4. Gestion d'exceptions (≈ 1 cours)

Une heure 

: à repartir dans la liste ci-dessus pour renforcer les notions compliquées

Les ressources du cours (diapos, tutos) :
https://gitlabinfo.iutmontp.univ-montp2.fr/dev-objets/ressources

TP

- 10-11 TPs différents
- Des tests automatiques pour valider vos solutions
- TP1 : https://gitlabinfo.iutmontp.univ-montp2.fr/dev-objets/TP1
- Essayez de travailler par vous-même : Copilot et ChatGPT ne seront pas là aux interros...

Enseignants :

malo.gasquet@umontpellier.fr - TP
sophie.nabitz@univ-avignon.fr - TP
cyrille.nadal@umontpellier.fr - TP (à Sète)
victor.poupet@umontpellier.fr - TP
gilles.trombettoni@umontpellier.fr - TP
petru.valicov@umontpellier.fr - CM/TP

Conseils pour mieux réussir

En TD c'est **VOUS** qui travaillez $\Longrightarrow$ Pas de correction à copier/coller $\Longrightarrow$ analyse/amélioration de **VOTRE** solution

- **N'apprenez pas par cœur, ça ne sert à rien !**
- Posez des questions en amphi (oui, vous pouvez toujours le faire)
- Posez des questions en TD
- Participez au forum du cours : https://piazza.com/class/lrahb0patze3u4
  - formuler une question permet de mieux comprendre le problème
  - généralement votre question intéresse plusieurs étudiants, ils n'ont juste pas osé la poser...
  - sur 6 enseignants, vous trouverez toujours un qui répondra rapidement
  - souvent les étudiants répondent aux questions
  - **très pratique pour le travail en distanciel**

Contrôle des connaissances

- **Petites interros** après certains TPs
- **Projet**
  - par équipes de 2 (les étudiants doivent être du **même groupe**)
  - période de réalisation : mars - début avril
  - notes individuelles en fonction de l'implication de chacun

Contrôle final

- Durée de 2-3 heures
- questions de cours + exercices
- documents autorisés : feuille A4 manuscrite recto-verso

Notation

SAE 1 et 2 : Projet découpé en plusieurs phases

Ressource : $\frac{1}{3} * \text{TP} + \frac{2}{3} * \text{contrôle final}$

Projet - *Trains*

- les règles :
  https://gitlabinfo.iutmontp.univ-montp2.fr/dev-objets/ressources
- commencez à réfléchir avec qui vous allez travailler en binôme
- **les consignes précises vous seront données ultérieurement** :
  - squelette du code avec batterie de tests obligatoires
  - modalités de rendu

Les outils

- Langage de programmation : `Java` ≥ 17
- La notation `UML`
- un IDE : `IntelliJ IDEA` ou autre
- Outil de build : `Maven`, version ≥ 3.8
- Les tests : `JUnit (version 5)`
- Versioning : `Git` et `GitLab`

...et pour commencer, au moins quelques principes de programmation :

KISS - **K**eep **I**t **S**imple and **S**tupid

YAGNI - **Y**ou **A**in't **G**onna' **N**eed **I**t

DRY - **D**on't **R**epeat **Y**ourself

Un peu de biblio

- H. Schildt. **Java : A Beginner's Guide, 7th Edition**
  McGraw-Hill : 2017 (6ème édition pour Java 8 disponible en ligne gratuitement)
- J. Blosch. **Effective Java, 3rd Edition**
  Addison-Wesley Professional : 2018.
- R.C. Martin.
  Clean Code - A Handbook of Agile Software Craftmanship
  Prentice Hall : 2008.
- E. Freeman, E. Robson, B. Bates, K. Sierra.
  Head First - Design Patterns (régulièrement mis à jour)
  O'Reilly : 2020.

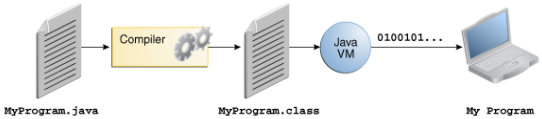
Pensez à consulter régulièrement la documentation officielle Java :

https://docs.oracle.com/en/java/javase/17/docs/api/index.html

Java

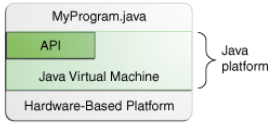
Java est un langage de programmation

- 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 ?

- 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.)

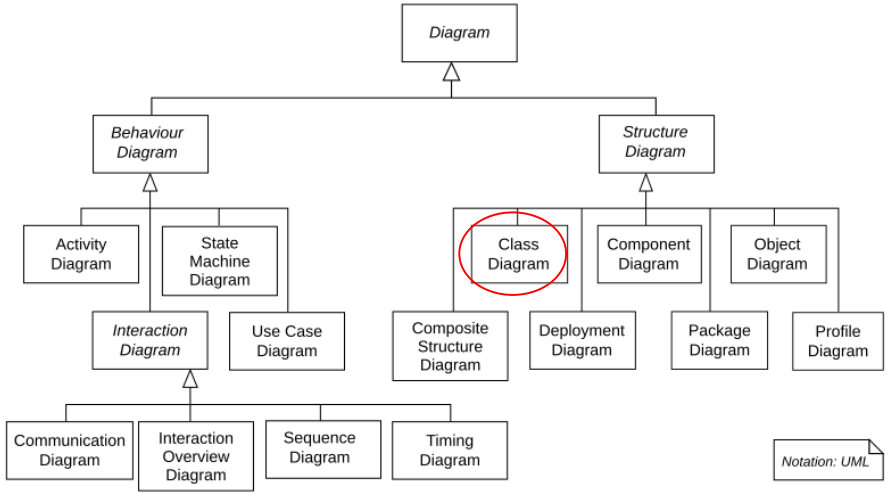
UML - Unified Modeling Language

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



Dans ce cours seuls les diagrammes de classes seront considérés.

Approches de programmation

Approche **structurelle (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.

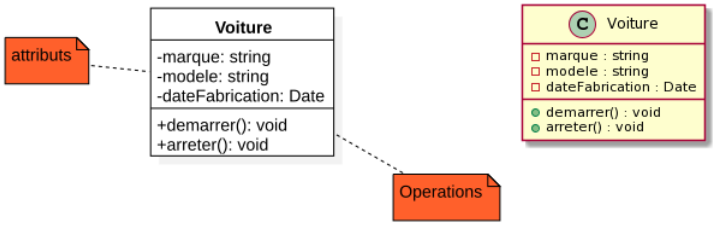
- La classe est un "**modèle**" pour la création de nouveaux objets.
- "**Contrat**" garantissant les compétences de ses objets.

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` :

Point2D
-x: double -y: double
+deplacer(x1: double, y1: double): void

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**.

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

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

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

}
```

```java
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 */
    }

}
```

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

```java
// 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 :

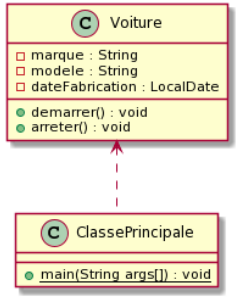
```
class Compte {
    private String nom, prenom;
    private double soldeCompte;

    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;
    }
}
```

Pensez à faire collaborer les constructeurs pour éviter la duplication de code !

Exemple d'une application orientée objet en 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 :



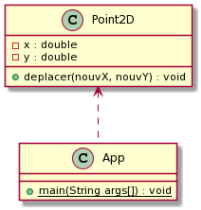

```
class ClassePrincipale {

    /* 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.

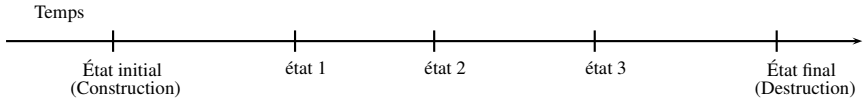
Petite recap – vocabulaire

Pour une classe :

- **attribut** = champs ≈ variable globale
- **fonction** = procédure = **méthode**
- **construction** = **instanciation** ≈ 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.

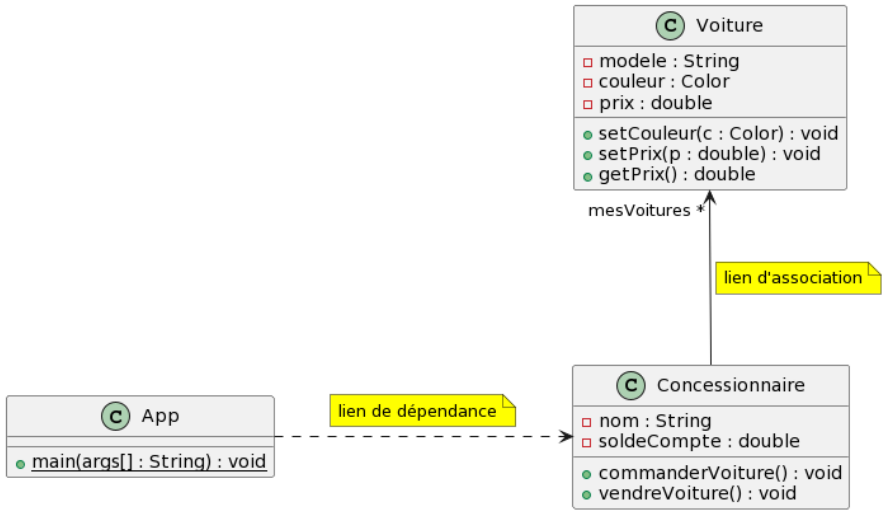
Cycle de vie des objets : exemple en Java

```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;
    }

    public double getPrix() {
        return prix;
    }
}
```

```java
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. */
    }
}
```

```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(){
        Voiture v = mesVoitures.get(0);
        soldeCompte += v.getPrix();

        // on va enlever de la liste la première
        // occurrence de la référence v
        mesVoitures.remove(v);
    }
}
```


Illustration en UML de l'exemple précédent



Classe–Type–Objet

Définition

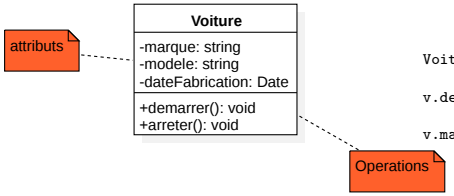
Un type de données définit :

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

```java
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)




```java
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 :

```java
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** :

```java
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
```

Types référence (objet)

- Hormis les types primitifs, tous les types en Java sont des objets.
- Exemples des types objets :
  - `String`
  - Tableaux, listes (et les autres structures de données)
  - Toutes les autres classes pré-définies en Java ou créées par l'utilisateur
- La valeur par défaut des types objets est `null` :

```
Voiture v;
System.out.println(v); // affiche "null"
```


Types primitifs *vs* types référence : Autoboxing

- En Java pour chaque type primitif il existe un type objet correspondant appelée *type enveloppe* (ou *wrapper type*)
  `Integer` pour `int`, `Boolean` pour `boolean`, *etc.*
- Les collections Java n'utilisent que des types objets :

```
ArrayList<int> liste = new ArrayList<>(); // ne compile pas
ArrayList<Integer> liste = new ArrayList<>(); // correcte
```

- Pour jongler entre les types enveloppes et les types primitifs, Java utilise le déballage/emballage automatique :

```
ArrayList<Integer> liste = new ArrayList<Integer>();
int nombre = 5;
liste.add(new Integer(nombre)); //emballage (boxing)
liste.add(nombre); //emballage automatique (autoboxing)
int v1 = liste.get(0); // déballage automatique (auto-unboxing)
int v2 = liste.get(1); // déballage automatique (auto-unboxing)
Integer v3 = liste.get(0);
Integer v4 = liste.get(1);
int v5 = v4.intValue(); // déballage
```


Utilité des types primitifs

L'esprit de la programmation orientée objet est que *tout* est objet.

à quoi bon avoir créé des types primitifs ? ? ?

Que fait-il ?

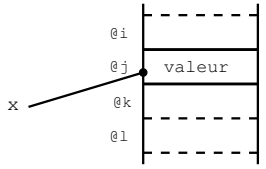
```
class App {
    public static void main(String args[]) {
       Integer somme = 0;
       for (int i = 0; i < 100000; i++)
          somme += i;

       System.out.println(somme);
    }
}
```

Inconvénients ?

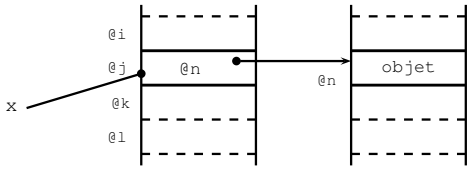
Typage Java : représentation en mémoire

Comment une variable `x` est représentée en mémoire ?

Cas 1 type primitif : `x = valeur`



Cas 2 type référence : `x = objet`



Accès par valeur *vs* accès par référence

```
int x = 10, y = 20;
x = y;
System.out.println(x + ", "+y); // resultat ?
y++;
System.out.println(x + ", "+y); // resultat ?
```

```
class Compte {

   private String nom, prenom;
   private double soldeCompte;

   public Compte(String n, String p) {
      nom = n; prenom = p;
      soldeCompte = 0;
   }

   public void rechargerCompte(double somme) {
      soldeCompte += somme;
   }


   // méthode spécifique Java pour l'affichage
   public String toString() {
      return "[" + prenom +
           ","+nom+","+soldeCompte +"]";
      }
}
```

```
Compte c1 = new Compte("Tartempion", "Riri");
c1.rechargerCompte(100);
Compte c2 = new Compte("Barbanchu", "Fifi");
c2.rechargerCompte(392);
Compte c3 = new Compte("Duchmolle", "LouLou");

System.out.println(c1); // resultat ?
System.out.println(c2); // resultat ?
System.out.println(c3); // resultat ?

c1 = c2;

System.out.println(c1); // resultat ?
System.out.println(c2); // resultat ?

c1.rechargerCompte(55);

System.out.println(c1); // resultat ?
System.out.println(c2); // resultat ?

c2 = c3;

System.out.println(c1); // resultat ?
System.out.println(c2); // resultat ?
System.out.println(c3); // resultat ?

c3 = c1;

System.out.println(c1); // resultat ?
System.out.println(c2); // resultat ?
System.out.println(c3); // resultat ?
```


Passage de paramètres

Dans les langages de programmation modernes, il existe essentiellement deux modes de passage de paramètres :

Passage par valeur

C'est *la valeur* de la variable au moment de l'appel qui est utilisée. Autrement dit, c'est une *copie* du contenu de cette variable au moment de l'appel qui est passée à la fonction.

⇒ La variable initiale n'est pas accessible (donc **non-modifiable**) par la fonction appelée.

Passage par adresse

C'est *l'adresse* de la variable placée en paramètre lors de l'appel qui est utilisée.

⇒ La variable initiale est accessible (et donc **modifiable**) par la fonction appelée

Passage de paramètres en Java

En Java, le passage de paramètres se fait toujours **par valeur**

Autrement dit, la méthode manipule une *copie* du contenu de la variable passée en paramètre.

1. si la variable est de type primitif : son contenu est une valeur effective, donc c'est en quelque sorte une "vraie" copie

```
public void toto() {
   int x = 10;
   System.out.println(x); // 10
   foo(x);
   System.out.println(x); // 10
}
```

```
public void foo(int v) {
   v = 30;
   // v est une copie de la variable passée en paramètre,
   // à la sortie de la fonction, les modifications
   // sur cette copie seront oubliées
}
```

2. si la variable est de type référence, alors la méthode manipule une copie de l'adresse (indiquant où est situé l'objet concerné). Donc : cette adresse est *non-modifiable* (car passage par valeur) **mais**

l'objet référencé, lui, est accessible, et donc modifiable à travers son interface publique.

Passage de paramètres en Java

Attention aux paramètres de type objet !

```
class Voiture {
   String modele;

   public Voiture(String m) {
      modele = m;
   }

   public void changerModele(String m) {
      modele = m;
   }

   public String toString() {
      return "La voiture est une "+modele;
   }
}
```

```
class Usine1 {
   public void customizer(Voiture v) {
      v = new Voiture ("berline");
   }
}
```

```
class Usine2 {
   public void customizer(Voiture v) {
      v.changerModele("berline")
   }
}
```

```
class App {
   /* Du code ici */
   public static void main(String args[]) {
      Voiture maCaisse = new Voiture("cabriolet");
      System.out.println(maCaisse); // resultat ?
      Usine1 u1 = new Usine1();
      u1.customizer(maCaisse);
      System.out.println(maCaisse); // resultat ?
   }
}
```

```
class App {
   /* Du code ici */
   public static void main(String args[]) {
      Voiture maCaisse = new Voiture("cabriolet");
      System.out.println(maCaisse); // resultat ?
      Usine2 u2 = new Usine2();
      u2.customizer(maCaisse);
      System.out.println(maCaisse); // resultat ?
   }
}
```

Référence this

En Java, le mot clef `this` permet de référencer l'objet *courant*.

```java
class Compte {

    private String nom, prenom;
    private double soldeCompte;

    public Compte(String n, String p) {
       nom = n;
       prenom = p;
       soldeCompte = 0;
    }

    public void rechargerCompte(double somme) {
       soldeCompte = somme;
    }

    // méthode spécifique Java pour l'affichage
    public String toString() {
       return "[" + prenom +
            ","+nom+","+soldeCompte +"]";
    }

}
```

```java
class Compte {

    private String nom, prenom;
    private double soldeCompte;

    public Compte(String nom, String prenom) {
       this.nom = nom;
       this.prenom = prenom;
       soldeCompte = 0;
    }

    public void rechargerCompte(double soldeCompte) {
       this.soldeCompte = soldeCompte;
    }

    // méthode spécifique Java pour l'affichage
    public String toString() {
       return "[" + prenom +
            ","+nom+","+soldeCompte +"]";
    }

}
```

C'est l'**identité** de l'objet.

Membres et méthodes static

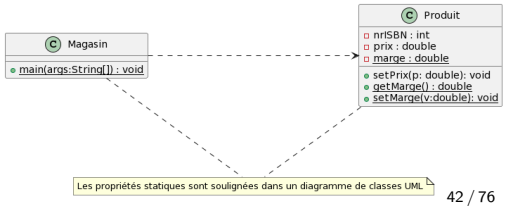
- Un attribut `static` est partagé par **toutes** les instances de la classe (valeur unique pour toutes ses instances)

```java
public class Produit {
    private int nrISBN;
    private double prix;

    private static double marge = 2.5;

    public Produit(int nrISBN, double prix) {
        this.nrISBN = nrISBN;
        this.prix = prix;
    }

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

    public static double getMarge() {
        return marge;
    }

    public static void setMarge(double v) {
        marge = v;
    }
}
```

```java
public class Magasin {
    public static void main(String args[]) {
        Produit tablette = new Produit(859587, 200);
        Produit pc = new Produit(465756, 952);

        // affiche 2.5
        System.out.println(Produit.getMarge());

        // on modifie la marge pour tous les produits
        Produit.setMarge(3.6);

        // affiche 3.6
        System.out.println(Produit.getMarge());
    }
}
```

Représentation en UML :

```java
tablette.setMarge(3.6); // instruction déconseillée !!!
```



Membres et méthodes static

Être déclaré en `static` est une contrainte très forte :

- aucune dynamique : pas de possibilité d'avoir une valeur d'attribut différente pour chaque objet de même type
- les méthodes `static` peuvent manipuler que des attributs `static`. Erreur du débutant : *déclarer tout en* `static`, *comme ça on n'en parle plus !* ☹
- la méthode `main(String)` de la classe principale est `static` car c'est le point d'entrée du programme
- on utilise souvent la clause `static final` pour déclarer des constantes :

  `public static final double PI = 3.14;`

Moralité

L'utilisation des membres `static` doit être justifiée.

Mot-clef final

Pseudo-définition

Une entité `final` ne peut être instanciée qu'une seule fois.

- pour une variable ou un attribut : une fois la valeur affectée, elle ne pourra plus être modifiée. Exemple :

```java
public void test1() {
   final int x = 10;

   System.out.println(x); // affiche 10

   x = 25; // ERREUR de compilation
   x++; // ERREUR de compilation
}
```

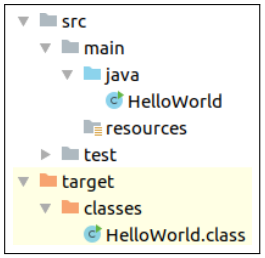
```java
public void test2() {
   final Voiture v = new Voiture("berline");
   v.setCouleur(Color.BLACK); // fonctionne car on change l'état
                              // mais pas l'objet lui-même

   v = new Voiture("berline"); // ERREUR de compilation
   v = new Voiture("cabriolet"); // ERREUR de compilation
}
```

- si la variable/attribut est une référence vers un objet, alors on peut modifier son état interne (à travers ses méthodes), mais la référence de l'objet sera la même
- les classes et les méthodes peuvent aussi être déclarées `final`
  - classe `final` : ne peut pas être "*sous-classée*" (à voir plus tard dans le cours)
  - méthode `final` : ne peut pas être "*redéfinie*" dans une "*sous-classe*"

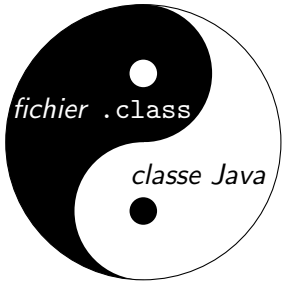
Environnement Java

De **point de vue physique** un programme Java est :

- un ensemble de fichiers sources `.java`
- un ensemble de fichiers compilés `.class` (le *byte-code*) correspondants aux sources
- le tout organisé dans des répertoires :
  - pensez à séparer le byte-code des sources



De **point de vue logique** un programme Java est un ensemble de classes organisées dans des *packages* (ou paquetages).



Environnement Java - les packages

Remarque

En Java chaque classe appartient nécessairement à un **package**.

- les packages permettent un découpage de l'espace de nommage en plusieurs sous-espaces de noms.
- un package est un ensemble cohérent de classes pouvant être importées toutes ensembles comme individuellement.
- package par défaut : *unnamed* (son utilisation est fortement déconseillée)
- le nom complet d'une classe (Fully Qualified Class Name ou FQCN) est constitué du nom du package auquel elle appartient suivi de son nom. Par exemple :

```
java.util.ArrayList // la classe ArrayList située dans le package java.util
```

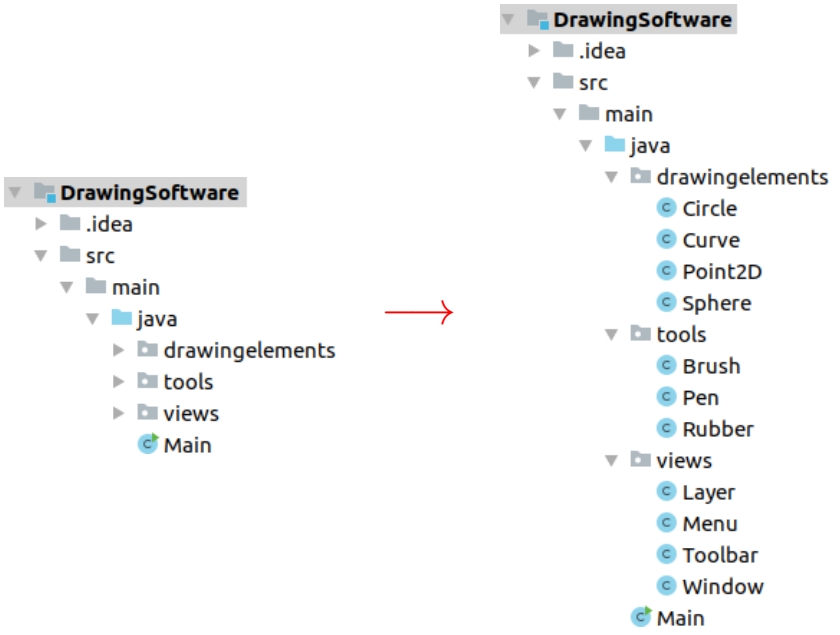

Environnement Java - les packages

```
// paquetage où est située la classe
package drawingElements

class Point2D {

    private double x;
    private double y;

    public Point2D(int x, int y){
       this.x = x;
       this.y = y;
    }

    public void move(double newX, double newY) {
       x = newX;
       y = newY;
    }

}
```

```
// paquetage où est située la classe
package drawingElements;

// classes importées dans le paquetage java.util
import java.util.ArrayList;

class Courbe {

    ArrayList<Point2D> listOfPoints;

    public Courbe(Point2D initialPoint){
       listOfPoints = new ArrayList<>();
       listOfPoints.add(initialPoint);
    }

    public void increase(Point2D newPoint) {
       listOfPoints.add(newPoint);
    }
}
```

- L'instruction "`package` `nomDuPackage`" doit être la première du fichier. Elle n'est pas obligatoire si le package est celui par défaut.
- Si deux classes **A** et **B** sont dans le même package, alors pas besoin d'importer quoique ce soit dans **B** pour accéder à la classe **A**.

Packages : illustration



Encapsulation

Une classe a deux aspects :

1. son *interface* : vue externe (ne pas confondre avec le mot-clé "`interface`" en Java)
2. son *corps* : implémentation des méthodes et des attributs

Définition

L'**encapsulation** est un principe de conception et de programmation consistant à distinguer l'interface et le corps d'une classe.

- un mécanisme de **visibilité** permet de contrôler les accès au corps d'un objet
- **seules** les méthodes doivent pouvoir manipuler l'état interne de l'objet et servent à la communication inter-objets

Principe fondamental de l'orienté objet

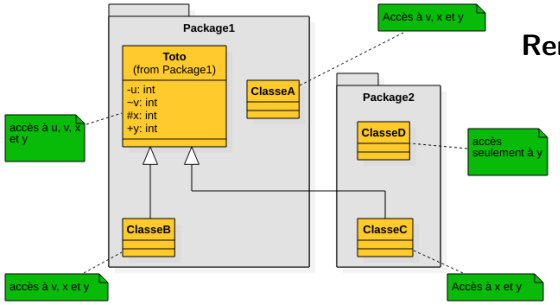
L'utilisateur/programmeur ne connaît que l'interface de l'objet. L'implémentation de sa classe est masquée et non accessible à l'utilisateur.

Encapsulation – visibilité

En Java, il y a 4 niveaux de visibilité :

```
class Toto {
   private int u; // seules les méthodes de la classe Toto y ont accès

   int v; // visibilité par défaut : accessible aux membres de la classe Toto
          // et de toutes les classes situées dans le même paquetage (package)

   protected int x; // accessible aux membres de la classe Toto, de ses sous-classes
                    //et de toutes les classes situées dans le même paquetage

   public int y; // accessible aux membres de toutes les classes
}
```

Notation en UML : $-$, $\sim$, $\#$, $+$



Remarques :

La notion de sous-classe sera vue ultérieurement.

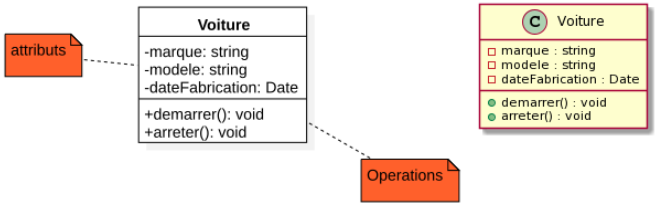
Ici `ClasseA` et `ClasseB` ont les mêmes accès aux membres de la classe `Toto`

Encapsulation : quelques conseils

- À priori, **tous les attributs** doivent être **privés**. L'utilisation de toute autre visibilité doit être argumentée.
  À votre avis, pourquoi ?
- Si besoin d'accéder à un attribut à partir d'une autre classe : créer un *accesseur* (ou *getter* en anglais).
- Si besoin d'accéder à un attribut à partir d'une autre classe : créer un *modifieur* (ou *setter* en anglais).
- Créer des accesseurs/modifieurs alors que l'on n'en a pas forcément besoin est un signe de non-respect du principe d'encapsulation.

Pensez au principe **YAGNI** (You Ain't Gonna' Need It) : protégez-vous de vos propres bêtises (et de celles des futurs collaborateurs)

Bilan UML jusque-là



- chaque classe est représentée graphiquement
- les méthodes et attributs "**essentiels**" sont illustrés
- les visibilité y sont représentées ($-$, $\sim$, $\#$, $+$)

Relations entre les classes vues à ce jour :

- **dépendance** : relation unidirectionnelle exprimant une dépendance *sémantique* entre deux classes
- **association** : relation sémantique entre les **objets** d'une classe

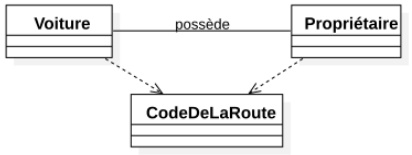
Dépendance

- Relation unidirectionnelle exprimant une dépendance sémantique entre deux classes
- Représenté par un trait discontinu orienté

- Généralement `A` dépend de `B` (on dit aussi `A` *utilise* `B`) si :
  - `A` utilise `B` comme argument dans la signature d'une méthode
  - `A` utilise `B` comme variable locale d'une méthode

Exemple : la modification du code de la route a un impact sur

- l'attitude du conducteur
- des caractéristiques des voitures



Relation très générale : par définition toutes les relations possibles entre les classes sont des dépendances

Relations entre classes : association

- relation sémantique entre les **objets** d'une classe
- peut être unidirectionnelle ou bidirectionnelle
- possède un *rôle* à chaque extrémité
  - décrit comment une classe voit une autre classe à travers l'association
  - **devient le nom d'un attribut en** `Java`
- multiplicité ou cardinalité

Une possible implémentation en `Java` :

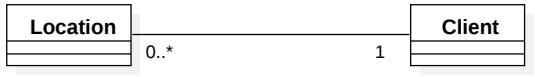
```java
class Personne {
    private Entreprise employeur;
}
```

```java
class Entreprise {
    // un tableau d'employés
    private ArrayList<Personne> employés;
}
```


Multiplicités des associations

La notion de **multiplicité** (ou **cardinalité**) permet de contraindre le nombre d'objets intervenant dans les instanciations des associations.

Exemple : *une location est payée par un et un seul client, alors que le client peut réserver plusieurs locations.*



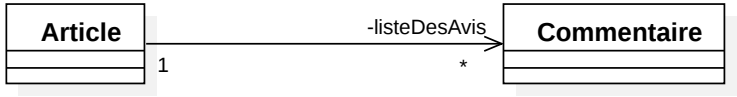
La syntaxe de m

- **1** : toujours un et un seul (dès la création de l'objet)
- **0..1** : zéro ou un
- `m..n` : de *m* à *n* (entiers > 0)
- ***** ou **0..*** : de zéro à plusieurs
- **1..*** : au moins un

} tableau/liste en `Java`

Navigabilité d'une association

- La **navigabilité** permet de spécifier dans quel(s) sens il est possible de traverser l'association à l'exécution.
- On restreint la navigabilité d'une association à un seul sens à l'aide d'une flèche.



Exemple : Connaissant un article on connaît les commentaires, mais pas l'inverse. En `Java`, `listeDesAvis` pourrait être un tableau/liste/collection de références vers des objets de type `Commentaire`.

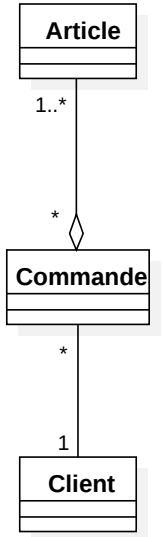
Une possible implémentation en `Java` :

```java
public class Article {
    private ArrayList<Commentaire> listeDesAvis;
}
```

```java
public class Commentaire {

}
```

Associations spéciales : agrégation

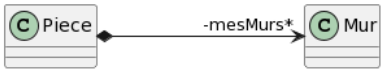
- Une **agrégation** est une forme d'association plus forte que l'association simple
- Représente la relation d'**inclusion faible** d'un élément dans un ensemble
- On représente l'agrégation par l'ajout d'un losange vide du côté de l'agrégat
- On utilise souvent le terme **composition faible**



Associations spéciales : composition

- L'association la plus "forte" : un losange plein
- Décrit une **contenance** structurelle entre instances
- Cardinalité maximum de 1 obligatoire

La **creation**/**destruction** du composant dépend entièrement de l'objet composite.



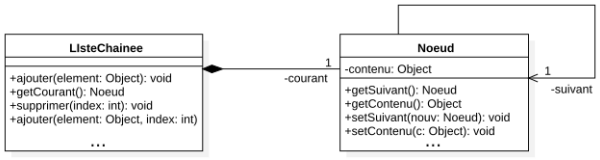
Dans cet exemple, les murs n'appartiennent qu'à une seule pièce...

Une implémentation possible Java :

```java
public class Piece {
   private ArrayList<Mur> mesMurs;

   public void construire() {
       Mur m = new Mur(); // instanciation des murs
       mesMurs.add(m);
   }

   // dans ce code, le cycle de vie des murs
   // dépend entièrement du cycle de vie de la pièce
}
```

```java
public class Mur {
}
```


Relations entre classes : association




```java
public class ListeChainee {
   private Noeud courant;

   public void ajouter(Object element) {
       Noeud nouveau = new Noeud(element);
       if (courant == null)
           courant = nouveau;
       else {
           Noeud tmp = courant;
           while (tmp.getSuivant() != null)
               tmp = tmp.getSuivant();
           tmp.setSuivant(nouveau);
       }
   }

   public Noeud getCourant() { return courant; }

   public void supprimer(int index) { ... }

   public void ajouter(Object element, int index) { ... }

   /* d'autres attributs et méthodes */
}
```

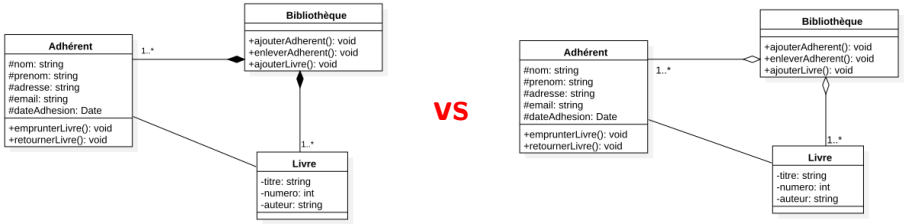
```java
public class Noeud {
   private Noeud suivant;
   private Object contenu;

   // constructeur
   public Noeud(Object contenu) {
      this.contenu = contenu;
   }

   public Noeud getSuivant() { return suivant; }

   public Object getContenu() { return contenu; }

   public void setSuivant(Noeud nouv) {
       suivant = nouv;
   }

   public void setContenu(Object c) {
       contenu = c;
   }

   /* d'autres attributs et méthodes */
}
```


associations spéciales : composition vs agrégation

- Dès que il y a la notion de contenance on utilise une agrégation ou une composition
- La composition est aussi dite **agrégation forte**

Comment décider entre la composition et l'agrégation ?

Si les composants ont une autonomie vis-à-vis du composite alors préférez l'agrégation. Mais tout dépend de l'application que vous développez...



Type énuméré

Parfois il est utile d'utiliser un ensemble prédéfini de constantes :

- les points cardinaux : nord, est, sud, ouest
- les 7 jours de la semaine
- les marges dans un document

On utilise les **types énumérés** dès lors que le domaine de valeur est *un ensemble fini et relativement petit de constantes* :

```
public enum Jour {
    LUNDI, MARDI, MERCREDI, JEUDI,
    VENDREDI, SAMEDI, DIMANCHE
}
```

```
public enum Coordonnee {
    NORD, EST, SUD, OUEST
}
```

```
public enum Margin {
    TOP, RIGHT, BOTTOM, LEFT
}
```

La déclaration `enum` définit une classe et donc on peut y ajouter des méthodes et autres champs.
Avantage principal : des noms claires/explicites à l'utilisation

Pour plus d'infos : https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

Type énuméré

```
public enum Jour {
    LUNDI, MARDI, MERCREDI, JEUDI,
    VENDREDI, SAMEDI, DIMANCHE
}
```

```
public static void main(String[] args) {
    System.out.println(Jour.MERCREDI);

    Jour j = Jour.MARDI;

    Commande c = new Commande(j);

    System.out.println(c.commanderRepas());
}
```

```
public class Commande {
    Jour date;

    public Commande(Jour date) {
        this.date = date;
    }

    public String commanderRepas() {
        switch (date) {
            case LUNDI:
                return "Sandwich";
            case MARDI:
                return "Pizza";
            case MERCREDI:
                return "Quiche";
            case JEUDI:
                return "Soupe";
            case VENDREDI:
                return "Salade";
            default:
                return "Pâtes";
        }
    }
}
```

Malgré le nombre limité de valeurs possibles, les `enum` sont des classes donc peuvent avoir d'autres attributs et méthodes.

Type `record`

Parfois il est utile de créer des objets **immuables** *i.e.* les objets dont l'état initial ne pourra jamais être modifié.

```
// une classe immuable
public final class Voiture {
    private final String immatriculation;
    private final int nbChevaux;

    public Voiture(String immatriculation, int nbChevaux) {
        this.immatriculation = immatriculation;
        this.nbChevaux = nbChevaux;
    }

    public String getImmatriculation() { return immatriculation; }

    public int getNbChevaux() { return nbChevaux; }

    public String toString() {
        return "Voiture[immatriculation=" + immatriculation + ", nbChevaux=" + nbChevaux + ']';
    }
}
```

```
// Utilisation
public static void main(String[] args) {
    Voiture v = new Voiture("QJ948XA", 120);
    System.out.println(v.getImmatriculation());
    System.out.println(v.getNbChevaux());
}
```

Exemples que vous connaissez :

- `java.lang.String`
- `Integer`, `Double`, ...
- `java.lang.LocalDate`

Type `record`

À partir de Java 16, la déclaration des types immuables a été facilitée avec l'introduction des `record` :

```
// équivalent à la classe Voiture précédente
public record Voiture(String immatriculation, int nbChevaux) { }
```

```
// utilisation d'un record
public static void main(String[] args) {
    Voiture v = new Voiture("QJ948XA", 120);
    // PAS DE v.getImmatriculation()
    System.out.println(v.immatriculation());
    // PAS DE v.getNbChevaux();
    System.out.println(v.nbChevaux());
}
```

Avantages :

- syntaxe plus succincte est claire
- possibilité d'imposer par contrat l'immutabilité d'une classe sans possibilité d'y déroger :

```
public record Voiture(String immatriculation, int nbChevaux) {
    private int prix ; // ERREUR DE COMPILATION
}
```

Écrire du code : règles de bon sens

Qu'est-ce que c'est ? ? ? ⟶

```
public class P {
  private C c;
  private List<A> cnt = new ArrayList<A>();

  public P(C c){
     this.c = c;
  }

  public void m(A a, int q) {
     for (int i = 0; i < q; i++) {
        cnt.add(a);
     }
  }

  public int ct() {
     int t = 0;
     for (A a : cnt) {
        t += a.getP();
     }
     return t;
  }
}
```

- programme claire ?
- comprend-on son but ? ou a-t-il besoin des commentaires supplémentaires ?

Écrire du code : règles de bon sens

Règle d'or

Donner des noms simples et descriptifs aux classes, attributs et méthodes.

```
public class P {
  private C c;
  private List<A> cnt = new
        ArrayList<A>();

  public P(C c){
     this.c = c;
  }

  public void m(A a, int q) {
     for (int i=0; i<q; i++) {
        cnt.add(a);
     }
  }

  public int ct() {
     int t = 0;
     for (A a: cnt) {
        t += a.getP();
     }
     return t;
  }
}
```

```
public class Panier {
  private Client client;
  private List<Article> contenu = new ArrayList<Article>();

  public Panier(Client client) {
     this.client = client;
  }

  public void ajouter(Article a, int quantite) {
     for (int i=0; i<quantite; i++) {
        contenu.add(a);
     }
  }

  public int calculerTotal() {
     int total = 0;
     for (Article a: contenu) {
        total += a.getPrix();
     }
     return total;
  }
}
```


Écrire du code : règles de bon sens

```
public void copier(int[] t1, int[] t2) {
    for (int i = 0; i < t2.length; i++) {
        t1[i] = t2[i];
    }
}
```

↘

```
public void copierTableaux(int[] t1, int[] t2) {
    for (int i = 0; i < t2.length; i++) {
        t1[i] = t2[i];
    }
}
```

↙

```
public void copierTableaux(int[] destination, int[] source) {
    for (int i = 0; i < source.length; i++) {
        destination[i] = source[i];
    }
}
```


Écrire du code : règles de bon sens

- 80% de la vie d'une application c'est de la maintenance et le code ne sera pas forcément maintenu par le même développeur
- Le lecteur doit comprendre le **sens** de votre code... sans avoir à comprendre les subtilités techniques/algorithmiques
- Essayez de faire des fonctions courtes et simples (une seule action à la fois)
- "*Don't comment bad code – rewrite it !*"

  B.W. Kernighan et P.J. Plaugher
- Testez, Relisez ⟶ Refactorisez (utilisez l'IDE)
- Plus de conseils dans
  Clean Code : A Handbook of Agile Software Craftsmanship de Robert C. Martin

Respectez les **conventions de nommage** du langage.

Conventions de nommage Java - sémantique

Les conventions ne sont pas imposées par le compilateur, mais tout le monde s'attend à ce qu'elles soient suivies.

- **classe** :
  - groupe nominal au singulier
  - exemples : `Client`, `Reservation`, `CompteEpargne`
- **attribut** et **variable** :
  - groupe nominal singulier ou pluriel
  - des noms très courts **uniquement** pour les variables à la durée de vie très courte (compteurs de boucle ou autre)
  - utiliser uniquement des caractères de l'alphabet [A-Z], [a-z], [0-9]
- **méthode** :
  - doit comporter un verbe (en général à l'infinitif)
  - `fermer()`, `ajouterElement()`, `calculerSomme()`, `getContenu()`
- anglais ou français ? **À vous de choisir mais pas de mélange** !

Conventions de nommage Java - lexique

- **package**
  - tout en minuscule avec les mots collés
  - les caractères autorisés sont alphanumériques (de a à z, de 0 à 9) et peuvent contenir des points
  - ex : `java.util`, `javafx.scene`, `fr.umontpellier.iut`
  - doit commencer par un nom de domaine (TLD) : `com`, `edu`, `org`, `fr`, etc.
- **classe** :
  - 1ère lettre en majuscule
  - mots attachés, avec majuscule aux premières lettres de chaque
  - ex. : `MaPetiteClasse`, `Voiture`, `ArrayList`
- **attribut**, **méthode** et **variable** : comme pour une classe mais 1ère lettre en minuscule
- **constante** (champ `static final`)
  - tout en majuscule avec des "_" entre les mots

`Fenetre.expand()` vs `fenetre.expand()`

Pour la première expression il s'agit d'un appel à une méthode statique

Qu'en pensez-vous ?

```java
import java.awt.Color;

public class Rectangle {
    private int longueur, largeur;
    private Color couleur;

    public Rectangle(int longueur, int largeur, Color couleur) {
        this.longueur = longueur;
        this.largeur = largeur;
        this.couleur = couleur;
    }

    public void changer(int x){
        longueur = x;
    }

    // on augmente suivant la proportion, mais on ne diminue jamais
    public void resize(int proportion){
        if (proportion > 1){
            longueur = longueur * proportion;
            largeur = largeur * proportion;
        }
    }

    public void changerCouleur(){
        couleur = Color.RED;
    }

    public Color coloration(){
        return couleur;
    }
}
```

Ça fonctionne mais...

Principe de Responsabilité Unique (SRP)

```java
public class Calculette {
  public static void additionner() {
      try (BufferedReader b = new BufferedReader(new InputStreamReader(System.in)))
      {
          System.out.println("Veuillez saisir le premier opérande : ");
          int x = Integer.parseInt(b.readLine());

          System.out.println("Veuillez saisir le second opérande");
          int y = Integer.parseInt(b.readLine());

          System.out.println("La somme est : " + (x+y));
      } catch (IOException e) {
          System.out.println("Erreur de saisie du nombre");
          System.out.println("Corrigez svp");
      }
  }
}
```

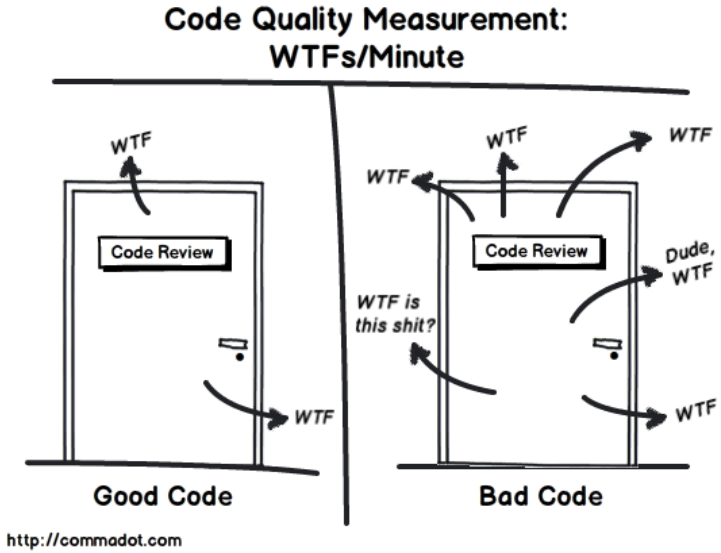
- s'il y a changement de formule d'addition ? – à priori c'est ok
- s'il y a changement de modalité d'impression ? – ça tient encore
- je veux ajouter la soustraction ! – il faut tout refaire → refactor

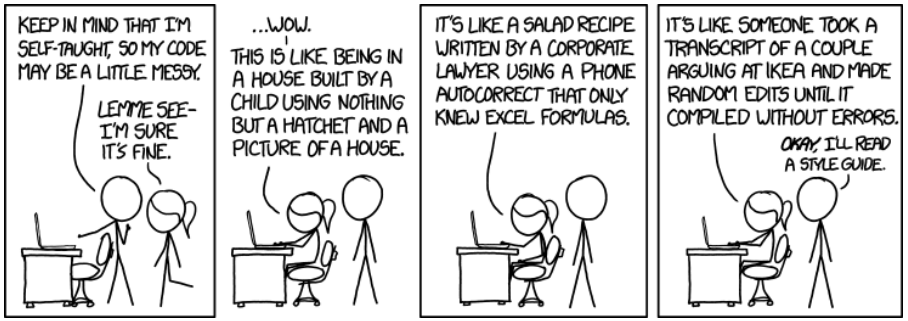
Principe de Responsabilité Unique (SRP)

```java
public class Calculette {
   public static int additionner(int x, int y) {
      return x + y;
   }

   public static int soustraire(int x, int y) {
      return x - y;
   }
}
```

```java
public class Imprimante {

   // les paramètres de configuration de l'imprimante

   public void imprimer(int valeur) {
      /* Préformatage de la valeur et
         impression correspondante */
   }
}
```

```java
public class Client {
   public static void main(Sring[] args) {
      try (BufferedReader b = new BufferedReader(new InputStreamReader(System.in)))
      {
         System.out.println("Veuillez saisir le premier opérande : ");
         int x = Integer.parseInt(b.readLine());
         System.out.println("Veuillez saisir le second opérande");
         int y = Integer.parseInt(b.readLine());
      }
      catch (IOException e) {
         System.out.println("Erreur de saisie du nombre");
         System.out.println("Corrigez svp");
      }

      int resultat = Calculette.additionner(x, y);
      Imprimante i = new Imprimante();
      i.imprimer(resultat);
   }
}
```



https://xkcd.com/1513/