

# L3. Programmation orientée objet. Cours 3

Marie-Pierre Béal (Cours de Rémi Forax)

Classes

Classes

# Classes

# Avant propos

Une des idées de Java est qu'une personne sur mille (le mainteneur) s'embête à écrire une librairie qui va aider les 999 autres (les utilisateurs) à écrire leurs codes facilement.

Une librairie est un jar (un zip glorifié) qui contient des fichiers classes (les .class).

Le site Maven Central contient

- 7 millions de jars
- plus de 400 000 artefacts

Le site est toujours en croissance exponentielle.

# Compatibilité descendante

Pour éviter d'avoir à ré-écrire le code utilisateur à chaque nouvelle version

- Java demande la compatibilité descendante binaire (binary backward compatibility)
- On a le droit d'ajouter de nouvelles fonctionnalités mais on ne doit pas casser les anciennes

Le langage sépare l'API publique

- accessible par les utilisateurs

de l'implantation

- accessible uniquement par les mainteneurs

# Classes et membres d'une classe

*Java manages to succeed, despite having almost all the defaults wrong.*

— *Brian Goetz*

# Membres d'une classe

Une classe contient

- Des champs d'instance ou statiques
  - cases mémoire contenant une valeur
- Des constructeurs (toujours d'instance)
  - méthodes d'initialisation des champs
- Des méthodes d'instance ou statiques
  - Code à exécuter en fonction des paramètres et des valeurs des champs

Quand vous serez plus grands, on pourra aussi mettre des classes dans les classes mais c'est compliqué.

# Visibilité

Les champs, constructeurs et méthodes peuvent être public ou private

- S'ils sont public, alors ils font partie de l'API

Il y a d'autres visibilités en Java,

- la visibilité de package (quand on écrit rien)
- la visibilité protected

mais on verra plus tard car leurs cas d'utilisation sont plus confidentiels.

Champs

Champs

# Champs à initialisation unique

Les champs peuvent

- être initialisés une seule fois par le constructeur (**final**)
- changer de valeur plusieurs fois

Utiliser des champs **final** rend le code plus maintenable, plus facile à débogger.

- si un champ final n'a pas la bonne valeur, alors la valeur envoyée au constructeur n'est pas la bonne

# Initialisation des champs

## Un champ

- **final** doit être initialisé dans le constructeur
- **pas final** ne doit pas forcément être initialisé dans le constructeur. Dans ce cas, il est initialisé à une valeur par défaut :
  - null pour les objets, 0 pour les entiers, 0.0 pour les doubles, false pour les booléen, etc.

Attention à ne pas confondre les champs (les cases mémoires des classes) et les variables locales (les cases mémoires des méthodes). Une variable locale doit toujours être initialisée.

## Exemple de code compliqué

```
public class Person {
    private /*pas final*/ String name;
    private /*pas final*/ int age;

    public Person(String name, int age) {
        // le constructeur est FAUX
        this.name = name;    // cf plus tard
        this.age = age;
    }

    public void updateAge() {
        this.age++;
    }
}
```

## Exemple plus simple

```
public class Person {
    private final String name;
    private final int age;

    public Person(String name, int age) {
        // le constructeur est FAUX
        this.name = name; // cf plus tard
        this.age = age;
    }
    public void updateAge() {
        this.age++; // ne compile pas !
    }
}
...
var person = new Person("Ana", 32);
doSomething(person);
// person.name et person.age n'ont pas changé
// pas besoin de regarder le code de doSomething()
```

## Faire des mutations

Et si on veut changer la valeur d'un objet

- On fait comme dans `String.toUpperCase()`, on renvoie un nouvel objet

Par exemple

```
public class Person {
    private final String name;
    private final int age;

    public Person(String name, int age) { ... }
    public Person updateAge() {
        return new Person(name, age + 1);
    }
}
```

# Classe immutable

Une classe qui a tous ses champs final est une classe dite non mutable ou immutable.

- String est non mutable
- StringBuilder est la version mutable de String
- Les records sont non mutables

Ces classes ont un comportement plus simple

- Rend le code plus facile à lire/débugger
  - Donc plus maintenable
- Mais changer une valeur entraîne une allocation
  - Les GCs de Java sont prévus pour cela (cas où les objets meurent vite)

Champs statiques

# Champs statiques

# Champ static final

Un champ **static final** est une constante

```
public record Asset(long price) {  
    private static final long MAX_TAX = 1_000_000L;  
  
    public long computeTax() {  
        return Math.min(MAX_TAX, price / 10);  
    }  
}
```

Aide à la lecture du code en remplaçant une valeur par un nom

- Aide à la maintenance du code

Constructeurs

# Constructeurs

# Constructeur

Un constructeur est une méthode d'instance qui initialise les champs

- Ayant le même nom que la classe
- Sans type de retour (c'est toujours void)
- Le premier paramètre est this (souvent implicitement)

On ne peut pas créer un objet sans appeler un constructeur (point d'entrée obligatoire)

- Donc le constructeur doit vérifier que l'on ne crée pas des objets faux (par exemple, une personne avec un age négatif)
- Ceci aide à la maintenance

# Préconditions

On appelle préconditions l'ensemble des conditions à vérifier pour que l'objet ne soit pas faux.

```
public class Person {
    private final String name;
    private int age;

    public Person(String name, int age) { // this est implicite
        Objects.requireNonNull(name, "name is null");
                                           // précondition

        if (age < 0) {
            throw new IllegalArgumentException("age < 0");
                                           // précondition
        }
        this.name = name;
        this.age = age;
    }
}
```

# Constructeur généré

Le compilateur ajoute automatiquement un constructeur

- pour un record, si aucun constructeur canonique (qui initialise tous les champs) n'est défini, un constructeur canonique est ajouté
- pour une classe, si aucun constructeur n'est défini, un constructeur public sans paramètre est ajouté par le compilateur

# Initialisation des champs à la déclaration

On peut initialiser des champs à la déclaration avec '='

```
public class Garage {  
    private final ArrayList<Car> cars = ...  
}
```

Le code du '=' est recopié au début de tous les constructeurs

```
public class Garage {  
    private final ArrayList<Car> cars;  
  
    public Garage() {  
        this.cars = ...  
    }  
}
```

# Surcharge de constructeurs

La "surcharge" de constructeur est le fait d'avoir plusieurs constructeurs.

- Ils doivent avoir des paramètres de types différents
- On utilise `this(...)` pour appeler un autre constructeur

Astuce pour avoir des valeurs par défaut

```
public class Car {
    private final String color;
    public Car(String color) {
        this.color = Objects.requireNonNull(color);
        // precondition
    }
    public Car() {
        this("red"); // appel Car(String)
    }
}
```

# Surcharge de constructeurs

La surcharge de constructeur est une pratique controversée.

- Du point de vue de l'utilisateur lors de l'écriture du code, cela veut dire qu'il faut faire un choix, donc lire la doc des multiples constructeurs
- Du point de vue du debugging, cela veut dire que l'on peut créer un objet avec des arguments cachés (les valeurs par défaut) ce qui n'aide pas à la compréhension du code

On préfère souvent avoir une façon unique de créer une instance d'une classe.

Méthodes d'instance

# Méthodes d'instance

## Méthode d'instance

Une méthode d'instance est une méthode dont le premier paramètre est `this` (peut-être implicitement).

On a donc besoin d'une instance pour pouvoir l'appeler

```
public class CarRental {
    ...
    public void rent(CarRental this) { ...
        // ici this est explicite
    }
}
...
var rental = new CarRental(...);
rental.rent() // appel la méthode rent()
              // avec rental en premier argument
```

# Méthode publique et préconditions

Comme pour les constructeurs, les méthodes publiques doivent tester les préconditions

```
public class MutablePerson {
    private final String name;
    private /*pas final*/ int age;
    private static int requireAgePositive(int age) {
        // Ne doit pas être public, c'est une méthode utilitaire
        if (age < 0) {
            throw new IllegalArgumentException("age < 0");
        }
        return age;
    }
    public MutablePerson(String name, int age) {
        this.name = Objects.requireNonNull(name);
                                                // préconditions
        this.age = requireAgePositive(age);
    }
    public void setAge(int age) {
        // attention, il faut aussi vérifier l'âge ici !!!
        this.age = requireAgePositive(age); // precondition
    }
}
```

# Surcharge de méthode d'instance

Java permet d'avoir plusieurs méthodes avec le même nom si leurs suites de types des paramètres sont différentes.

- Utilisé par `PrintStream` (car Java n'a pas un type commun pour les objets et les primitifs)
  - `void println(int)`
  - `void println(Object)`
  - `void println(String)`
  - ...

Même problème que pour les constructeurs, on évite de trop utiliser la surcharge en pratique

Méthodes statiques

# Méthodes statiques

# Méthode statique

Une méthode statique est une méthode que l'on appelle sur la classe indépendamment d'une instance

```
public class Car {  
    private Car(...) { // on empêche de créer une instance  
        ...           // sans passer par loadCarFromFile  
    }  
    public static Car loadCarFromFile(Path path) {  
        // lit le fichier et crée une instance  
        return new Car(...);  
    }  
}  
...  
var car = Car.loadCarFromFile(...);
```

Cela permet d'exécuter du code **avant** d'appeler le constructeur.

`toString()`, `equals()` et `hashCode()`

`toString()`, `equals()`  
et `hashCode()`

## toString()

Méthode appelée automatiquement par un `PrintStream` (comme `System.out` ou `System.err`) pour transformer un objet en `String` en vue de l'afficher

```
var object = ...  
System.out.println(object); // appel object.toString()
```

Si on veut son propre affichage, il faut redéfinir/remplacer la méthode `toString()`

```
public record Author(String name, long books) {  
    @Override  
    public String toString() { // remplace le toString() existant  
        return name + " " + books;  
    }  
}
```

L'annotation `@Override` demande au compilateur de vérifier que la méthode que l'on veut remplacer existe bien

## equals(), hashCode() et record

JDK possède déjà des structures de données, liste, table de hachage, etc. Celles-ci demandent que equals() et hashCode() soient implantées sur les éléments.

Un record implante automatiquement equals() et hashCode()

```
public record Author(String name, long books) {...}
...
var list = List.of(new Author("JRR Tolkien", 13));
list.contains(new Author("JRR Tolkien", 13)) // true
```

## equals(), hashCode() et classe

Contrairement à un record, une classe **n'implante pas** equals/hashCode correctement (il faut implanter les deux!!)

```
public class Author {
    private final String name;
    private final int books;
    public Author() { ... } // obvious code
}
...
var list = List.of(new Author("JRR Tolkien", 13));
list.contains(new Author("JRR Tolkien", 13)) // false
```

# Implanter equals(Object)

Remplacer boolean equals(Object)

- Il faut que la méthode que l'on définit ait la même signature (même visibilité, mêmes paramètres, même type de retour)

```
public class Author {
    private final String author;
    private final int books;
    ...
    @Override
    public boolean equals(Object o) {
        // Attention : ne prend pas un Author en paramètre
        // vérifier que 'o' est bien un Author et
        // tester les champs avec == (primitif)
        // ou equals (objet)
    }
}
```

## Écrire equals(Object)

L'opérateur `instanceof` permet de tester à l'exécution si l'Object pris en paramètre est bien un Author

```
public class Author {
    private final String name;
    private final int books;
    ...
    @Override
    public boolean equals(Object o) {
        return o instanceof Author author
            && books == author.books
            && name.equals(author.name);
        // On teste le primitif d'abord car == est plus rapide
        // que equals() et && est paresseux
    }
}
```

L'opérateur `instanceof` renvoie vrai et remplit la variable (“author”) si l'objet est bien un Author à l'exécution, ou renvoie false sinon

## Implanter hashCode()

Si deux objets sont égaux au sens de equals()

- `o1.equals(o2) == true`

alors ils doivent avoir la même valeur de hashCode()

- `o1.hashCode() == o2.hashCode()`

```
public class Author {
    private final String author;
    private final int books;
    ...
    @Override
    public boolean equals(Object o) {
        // Attention : ne prend pas un Author en paramètre
        // vérifier que 'o' est bien un Author et
        // tester les champs avec == (primitif)
        // ou equals (objet)
    }
}
```

## Ecrire hashCode()

Il y a deux façons de combiner des hashCodes

- Si on a deux valeurs, on utilise  $\wedge$  (le "ou exclusif") entre les deux hashCode
- Si on a plus de deux valeurs, on utilise `java.util.Objects.hash()`

```
public class Author {
    private final String name;
    private final int books;
    ...
    @Override
    public int hashCode() {
        return name.hashCode() ^ Integer.hashCode(books);
        // return Objects.hash(name, books);
        // marche aussi mais plus lent
    }
}
```

## Et si on n'implante pas equals correctement

Si on écrit equals(Author) au lieu de equals(Object) et on oublie le @Override

```
public class Author {
    private final String name;
    private final int books;
    ...
    public boolean equals(Author author) { ...
    }
}
```

Le code compile (ahh) mais ne marche pas correctement.

- Pour Java, il y a deux méthodes equals : equals(Object) qui est toujours là et equals(Author). Donc il y a surcharge et pas redéfinition.
  - equals(Author) ne sera jamais appelée car le code de java.util appelle equals(Object).
  - Toujours mettre @Override qui vérifie que l'on remplace bien la méthode.

## En mettant tout ensemble

Si on veut qu'une classe puisse être utilisée dans les structures de données prédéfinies de Java, il faut écrire equals et hashCode (les deux!!)

```
public class Car {
    private final String color;
    private final int seats;
    private final boolean fancy;
    ... // obvious constructor
    @Override
    public boolean equals(Object o) {
        return o instanceof Car car && fancy == car.fancy
            && seats == car.seats && color.equals(car.color);
    }
    @Override
    public int hashCode() {
        return Objects.hash(color, seats, fancy);
    }
}
```



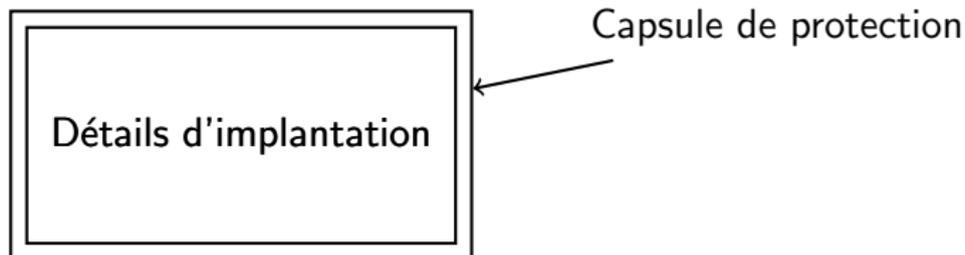
Encapsulation

# Encapsulation

# Encapsulation

L'encapsulation, c'est le fait de cacher les détails d'implantation pour faciliter la maintenance du code.

- Les utilisateurs de la classe voient l'API qui ne bouge pas
- Les mainteneurs de la classe changent l'implantation tout en restant compatible avec l'API



# Classe vs record

Un record ne permet pas l'encapsulation car les composants d'un record sont visibles par tous.

Contrairement à un record, une classe permet de séparer l'API et l'implantation

- record + encapsulation == classe

## Exemple

```
public record Point(int x, int y) { }
```

ou

```
public class Point {  
    private final int x;  
    private final int y;  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public int x() { return x; }  
    public int y() { return y; }  
    // + toString, equals et hashCode  
}
```

## Exemple

On veut avoir des coordonnées avec des flottants. Mais on ne veut pas changer tous les codes qui utilisent l'API.

```
Point point = ...  
int x = point.x();  
int y = point.y();  
doSomething(x, y);
```

## La mauvaise façon

```
public record Point(double x, double y) { }  
    // l'API n'est pas compatible
```

ou

```
public class Point {  
    private final double x;  
    private final double y;  
    public Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
    public double x() { return x; }  
    public double y() { return y; }  
    // + toString, equals et hashCode  
}
```

# La mauvaise façon

Le code ci-dessous ne compile plus, ahhhhh !

```
Point point = ...  
int x = point.x();  
int y = point.y();  
doSomething(x, y);
```

## La bonne façon

```
// public record Point(double x, double y) { }
public class Point {
    private final double x;
    private final double y;
    private Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public Point(int x, int y) {
        // on garde la même API publique
        this((double) x, (double) y);
    }
    public int x() { return (int) x; }
        // on garde la même API publique
    public int y() { return (int) y; }
        // on garde la même API publique
    // + toString, equals et hashCode
}
```

# La bonne façon

Le code ci-dessous marche maintenant, yeah !

```
Point point = ...  
int x = point.x();  
int y = point.y();  
doSomething(x, y);
```

# API ?

## Application Programming Interface

- Point de contact entre deux parties d'un programme
- Ensemble des méthodes d'utilisation d'une classe
  - permet de cacher les détails d'implantation

En Java, l'API est l'ensemble des "membres" public d'une classe publique

- constructeurs et méthodes publiques
- les champs ne sont jamais publiques (pas maintenable)

En résumé, en mémoire ?

En résumé, en mémoire ?

```

public class Car {
    static int DEFAULT_WHEELS = 42;
    String name;
    int wheels;
    public Car(...) { ... }
    public void m() { ... }
    private void m2() { ... }
    public static void m3() { ... }
}

```

```

String toString(Object this) { .. }
boolean equals(Object this, Object other) { ... }
int hashCode(Object this) { ... }

```

class  
wheels  
name



DEFAULT\_WHEELS

vtable

0:	
1:	
2:	
3:	
	42

void <init>(Car this, ...) { ... }

void m(Car this) { ... }

void m2(Car this) { ... }

void m3() { ... }

```

var car = new Car("reliant regal", 3);
    // car = new Car
    // &<init>(car, ...)
car.m(); // car.class[3](car)
car.m2(); // &m2(car)
Car.m3(); // &m3()

```

"reliant regal"

# Résumé

Une classe définit des champs (cases mémoire), un constructeur (point d'entrée d'initialisation) et des méthodes (fonctions liées à la classe)

## Une classe

- Utilise l'encapsulation (privée/publique)
- Est non mutable par défaut (champs final et private)
- Vérifie les préconditions dans les constructeurs publics et les méthodes publiques
- Ne modifie pas la signature des membres public d'une version à l'autre