

# ESIEE. Programmation orientée objet. Cours 3

Marie-Pierre Béal (Cours de Rémi Forax)

Records et Classes. Champs et méthodes statiques.



Records

Records

# Les records

On va écrire

- un record Point dans un fichier séparé de nom Point.java.
- une méthode statique main dans une classe Test.java.

```
// dans le fichier Point.java  
record Point(int x, int y) {}
```

# Les records

```
// dans Test.java
class Test.java {
    static void main() {
        var point = new Point(3, 4);
        IO.println(point);
    }
}
```

On peut compiler et exécuter avec

```
$ javac *.java
```

```
$ ls
```

```
Point.class Point.java Test.class Test.java
```

```
$ java Test
```

```
Point[x=3, y=4]
```

# Les champs

Les champs sont accessibles uniquement entre l'accolade ouvrante et l'accolade fermante du record. Ils ne sont pas accessibles en dehors du record. On dit que les champs sont **private**, qui veut dire non visible à l'extérieur.

```
// dans le fichier Point.java
record Point(int x, int y) {
    double distanceToOrigin() {
        return Math.sqrt(x * x + y * y);
    }
}
```

```
// dans Test.java
class Test.java {
    static void main() {
        var point = new Point(3, 4);
        IO.println(point.x); // ne compile pas
    }
}
```

## Un record public

```
// dans le fichier Point.java  
public record Point(int x, int y) {}
```

```
// dans Test.java  
class Test.java {  
    static void main() {  
        var point = new Point(3, 4);  
        IO.println(point.x()); // 3  
    }  
}
```

Ici on a ajouté un modificateur de visibilité **public** qui signifie que le record sera visible dans tout code situé dans n'importe quel répertoire.

# Accesseurs publics

Dans un record, le compilateur ajoute des méthodes publiques de même noms que les champs appelées accesseurs (ici `x()` et `y()`), avec comme code, le code des méthodes `x()` et `y()` généré par le compilateur. C'est comme si on écrivait

```
public record Point(int x, int y) {  
    public int x() { // il est inutile de l'écrire  
        return x;  
    }  
}
```

inutile signifie qu'il ne faut pas l'écrire (points en moins aux examens si vous l'écrivez).

# Remplacer / Redéfinir les méthodes existantes

On peut changer l'implantation des accesseurs ou des méthodes toString/equals/hashCode.

```
public record Pair(String first, String second) {  
    @Override  
    public String toString() {  
        return "Pair(" + first + ", " + second + ")";  
    }  
}
```

```
// dans Test.java  
class Test.java {  
    static void main() {  
        IO.println(new Pair("toto", "titi")); // Pair(toto, titi)  
    }  
}
```

On utilise l'annotation `@Override` pour aider à la lecture, faire la différence entre une nouvelle méthode et le remplacement d'une méthode existante.

## Accès implicite à this

Dans une méthode d'un record, l'accès à un champ ou une méthode n'a pas besoin d'être préfixé par `this` (instance) ou par le nom de la classe (static).

```
public record Point(int x, int y) {  
    public double distanceToOrigin() {  
        return Math.sqrt(sqr(x) + sqr(y));  
        // pas sqr(this.x) + ...  
    }  
    private static double sqr(int value) {  
        return value * value;  
    }  
}
```

# Constructeur canonique

Le constructeur est une méthode spéciale appelée lors du new pour initialiser les champs.

Dans un record, le compilateur génère automatiquement le constructeur canonique (celui qui initialise les champs).

```
public record Person(String name, int age) {  
    // généré automatiquement  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

# Redéfinir le constructeur

Il est souvent nécessaire de remplacer le constructeur car on veut empêcher de créer des objets avec des valeurs erronées

```
public record Person(String name, int age) {  
    public Person(String name, int age) {  
        Objects.requireNonNull(name);  
        // plante (lève une NullPointerException)  
        // si name est null  
        if (age < 0) {  
            throw new IllegalArgumentException("age < 0");  
        }  
        this.name = name;  
        this.age = age;  
    }  
}
```

On vérifie les pré-conditions.

# Redéfinir le constructeur compact

Le constructeur canonique a une version “compacte”

```
public record Person(String name, int age) {  
    public Person { // pas de parenthèses  
        Objects.requireNonNull(name, "name is null");  
        if (age < 0) {  
            throw new IllegalArgumentException("age < 0");  
        }  
    }  
}
```

qui ne laisse apparaître que les pré-conditions.

Le compilateur ajoute les `this.name = name;` etc à la fin du constructeur compact.

## equals/hashCode/toString

Dans un record, le compilateur génère aussi les méthodes

- `equals()` : indique si deux objets de type `Point` ont les mêmes valeurs

```
static void main(String[] args) {  
    var point = new Point(2, 3);  
    var point2 = new Point(2, 3);  
    println(point.equals(point2)); // true  
    var point3 = new Point(4, 7);  
    IO.println(point.equals(point3)); // false  
    IO.println(point.hashCode()) // 65  
}
```

- `hashCode()` : renvoie un entier "résumé", cf cours 4

## equals/hashCode/toString

- toString() : renvoie une représentation textuelle

```
static void main(String[] args) {  
    var point = new Point(2, 3);  
    IO.println(point.toString()); // Point[x = 2, y = 3]  
    IO.println(point); // Point[x = 2, y = 3]  
}
```

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

- 43 millions de jars
- des millions d'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*

# Classe

Dans un fichier HelloWorld.java on met :

```
public class HelloWorld {  
    static void main() {  
        IO.println("Hello World!");  
    }  
}
```

```
$ java HelloWorld.java  
Hello World!
```

# Classe

On définit ainsi une classe

- Le nom de la classe doit être le nom du fichier (sans le suffixe `.java`).
- Le nom de la classe (et donc du fichier) doit commencer par une majuscule.

Et lors de l'exécution, la méthode statique `HelloWorld.main()` est appelée.

# Classe compacte

En Java, si on définit des méthodes sans classe, le compilateur ajoute une classe autour avec le même nom que le nom du fichier. Une telle classe s'appelle une classe compacte.

Si on écrit dans un fichier HelloWorld.java

```
void main() {  
    IO.println("Hello World!");  
}
```

le compilateur génère

```
class HelloWorld {  
    public HelloWorld() { } // constructeur par défaut  
    void main() {  
        ....  
    }  
}
```

et lors de l'exécution `new HelloWorld().main()` est appelée

# 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 visibilité 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 débbuger.

- 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

Ne pas écrire une classe comme celle-ci

```
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)

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 âge 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 final 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

# 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.

Ne pas écrire de classe mutable comme celle-ci

```
public class MutablePerson {  
    private final String name;  
    private /*pas final*/ int age;  
  
    public MutablePerson(String name, int age) {  
        this.name = Objects.requireNonNull(name);    // preconditions  
        if (age < 0) {  
            throw new IllegalArgumentException("age < 0");    // précondition  
        }  
        this.age = age;  
    }  
    public void setAge(int age) {  
        // attention, il faut aussi vérifier l'age ici !!!  
        if (age < 0) {  
            throw new IllegalArgumentException("age < 0");    // précondition  
        }  
        this.age = age;  
    }  
}
```

# Surcharge de méthode d'instance

Java permet d'avoir plusieurs méthodes avec le même nom si leurs suites des 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

Module, Package et import

# Module, Package et import

# Package

Une librairie Java (appelé un module) est composée de plusieurs packages.

La librairie par défaut du langage Java est appelée `java.base` et contient les packages :

Pour le JDK (la librairie par défaut de Java)

- `java.lang` : classes de base du langage
- `java.util` : classes utilitaires, structures de données
- `java.util.regex` : expression régulière (cf cours 2)
- `java.io` : pour faire des entrées/sorties
- `java.nio.file` : entrées/sorties sur les fichiers
- etc.

# La directive import en Java

On peut faire une importation de tous les packages d'un module avec `import module` (JEP 476).

En début de fichier, on écrit :

```
import module java.base; // ne pas oublier le ';'
```

`import` n'importe pas de fichier au sens de Python/C mais dit que l'on peut utiliser `ArrayList` à la place de `java.util.ArrayList` dans le code. Le mot-clef devrait s'appeler "alias" pas "import".

Le `import module java.base` est fait automatiquement sur une classe non-nommée.

# La directive import en Java

Ou bien, on spécifie des classes/records appartenant à des packages que l'on veut utiliser

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

# Exemple

```
import module java.base;
public class Hello {
    static void main() {
        var list = new ArrayList();
    }
}
```

- Le vrai nom de ArrayList est java.util.ArrayList et le package java.util est importé car il est dans java.base.

Le code ci-dessus est équivalent au code sans import

```
public class Hello {
    static void main() {
        var list = new java.util.ArrayList();
    }
}
```

toString(), equals() et hashCode()

toString(), equals()  
et hashCode()

# toString()

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

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

## toString() et record

Rappel : dans un record, toString() est déjà implementé et si on veut son propre affichage, il faut redéfinir/remplacer la méthode toString()

```
public record Author(String name, int 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

## toString() et classe

Dans une classe, toString() donne un affichage par défaut qui en général ne convient pas. Il faut redéfinir/remplacer la méthode toString()

```
public class Author {
    private final String name;
    private final int books;
    ....
    @Override
    public String toString() { // remplace le toString() existant
        return name + " " + books;
    }
}
```

```
static void main() {
    var author = new Author("JRR Tolkien", 13);
    IO.println(author); // JRR Tolkien 13
    // sans redéfinition on aurait obtenu Author@702657cc
}
```

## `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, int 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'implémente pas** equals/hashCode correctement (il faut implémenter 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)  
    }  
    @Override  
    public int hashCode() { ...  
    }  
}
```

# 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  
    }  
    @Override  
    public int hashCode() { ...  
    }  
}
```

## Implanter hashCode() (le contrat objet)

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) {  
        ...  
    }  
    @Override  
    public int hashCode() {  
        ...  
    }  
}
```

## Écrire hashCode()

Il y a deux façons de combiner des hashCode

- 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 appelé 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);
    }
}
```

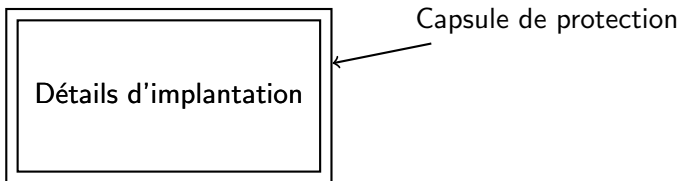
Encapsulation

# 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 == classe — encapsulation

# 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 publics
- les champs ne sont jamais publics (pas maintenable)

Static

static

Champs statiques

# Champs statiques

# Champ static

Un champ **static** est précédé du mot `static`. Il n'est pas stocké dans l'objet mais dans la classe.

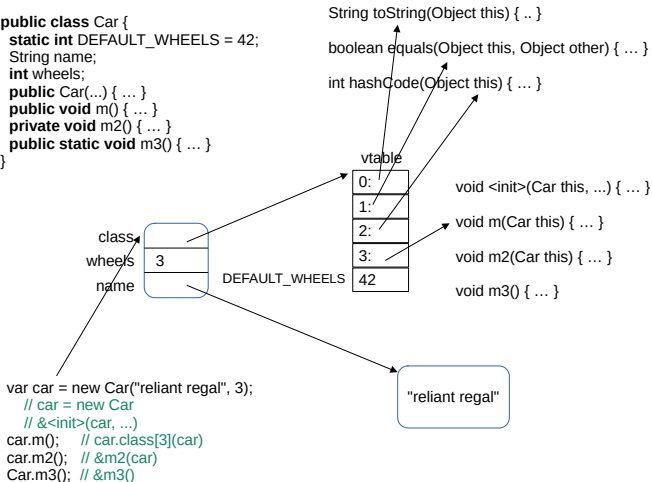
En mémoire ?

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() { ... }
}

```



# Exemples de champs statiques

La classe `System` a des champs statiques :

- `System.out`
- `System.err`
- `System.in`

```
static void main(){  
    System.out.println("Hello");  
}
```

## Champ statique non final (à ne pas faire)

Ne pas écrire un code comme celui là

```
public class Person {  
    private static String name;  
    public Person(String s) {  
        name = Objects.requireNonNull(s);  
    }  
    @Override  
    public String toString() {  
        return name;  
    }  
}
```

## Champ statique non final (à ne pas faire)

Ne pas écrire un code comme celui là

```
public class Test {  
    static void main(){  
        var bob = new Person("bob");  
        IO.println(bob);  
        var alice = new Person("alice");  
        IO.println(alice);  
        IO.println(bob);  
    }  
}
```

On obtient en sortie

bob  
alice  
alice

# 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

# Champ static pas final (à ne pas faire)

Un champ static qui n'est pas final est une sorte de variable globale partagée par la classe ou le record.

Il ne faut pas en mettre sauf cas exceptionnel.

Méthodes statiques

# Méthodes statiques

# Méthodes d'instance et méthodes statiques

On appelle

- une méthode d'instance sur une instance (avec un `.` après l'objet)
- une méthode statique sans instance, sur la classe (avec un `.` après **le nom de la classe**)

Lors de la déclaration

- une méthode statique est précédée du mot **static**.

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

# Méthodes d'instance et méthodes statiques

`java.io.IO.println` est une méthode statique

```
static void main(){  
    IO.println("Hello");  
}
```

Il existe aussi une méthode non statique `System.out.println()`

```
static void main(){  
    System.out.println("Hello");  
}
```

# Méthodes d'instance et méthodes statiques

`I0.readLine` et `Integer.parseInt` sont des méthodes statiques

```
static void main(){  
    var s = I0.readLine("Enter an integer \n");  
    I0.println(Integer.parseInt(s));  
}
```

```
$ java ParseIntTest.java
```

```
Enter an integer
```

```
234
```

```
234
```

# Méthodes d'instance et méthodes statiques

```
record Taxi(boolean uber) {  
    String name() { // this implicite  
        return this.uber? "Uber": "Hubert?";  
    }  
    static String bar() { // this n'existe pas ici !  
        return "Hello Taxi";  
    }  
}
```

```
static void main() {  
    IO.println(new Taxi(true).name()); // name méthode d'instance  
    IO.println(Taxi.bar()); // bar méthode statique  
}
```

```
$ java TaxiTest.java  
Uber  
Hello Taxi
```

## Et le main ?

Si on veut un main static, il faut le déclarer "static".

Dans un fichier HelloWorld.java on met :

```
static void main(){  
    IO.println("Hello World!");  
}
```

```
$ java HelloWorld.java  
Hello World!
```

appelle HelloWorld.main().

# Un main non statique

```
void main() {  
    IO.println("Hello World!");  
}
```

Ici la méthode `main()` n'est pas `static`.

C'est comme si on faisait un `new HelloWorld().main()`.

# Un main non statique qui ne marche pas

Ne pas écrire un code comme celui-là

```
public record Person(String name){  
    void main() {  
        IO.println("Hello");  
    }  
}
```

donne

Error: no non-private zero argument constructor found  
in class Persons remove private from existing constructor  
or define as: public Person()

# Exemples de méthodes statiques utilitaires

Les méthodes publiques doivent tester les préconditions.

Ne pas écrire de classe mutable comme celle-ci

```
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);    // preconditions  
        this.age = requireAgePositive(age);  
    }  
  
    public void setAge(int age) {  
        // attention, il faut aussi vérifier l'age ici !!!  
        this.age = requireAgePositive(age);    // precondition  
    }  
    this.age = age;  
}
```

# Méthode statique pour créer des objets

On rappelle qu'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.

# 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 (privé/public)
- Doit être écrite 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

# Résumé

Un record définit des champs (cases mémoire), un constructeur (point d'entrée d'initialisation) et des méthodes (fonctions liées au record)

Un record

- Ne permet pas l'encapsulation
- A ses champs tous privés et final mais il y a des accesseurs publics à ces champs
- A un constructeur canonique déjà défini mais qui ne vérifie pas les pré-conditions. Il faut écrire un constructeur compact pour vérifier les pré-conditions.
- A des méthodes `equals`, `hashCode`, `toString` déjà définies avec un bon comportement.

## Résumé : classes ou records, que choisir ?

- Si l'encapsulation n'est pas nécessaire, on prend un record. Par exemple pour définir un type qui a des champs non mutables (Point, Car, ...)
- Si un des champs est mutable, par exemple une liste modifiable (Garage, ...), on prend une classe. Attention, même si un champ liste est défini `private final`, l'accès à la liste permet d'ajouter des éléments dedans (voir cours suivants).