# Programmation orientée objet. Cours 3

Marie-Pierre Béal
UGE BUT 1

Classes, encapsulation et visibilité

# 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 nouvelle 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.

# 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

# 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 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 débugger.

- 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

# 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 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

# 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() {
    ...
  }
}
...
var rental = new CarRental(...);
rental.rent()  // appel la méthode rent()
               // avec rental en premier argument
```

# 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 hashCodes

- Si on a deux valeurs, on utilise ^ (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 ! !)

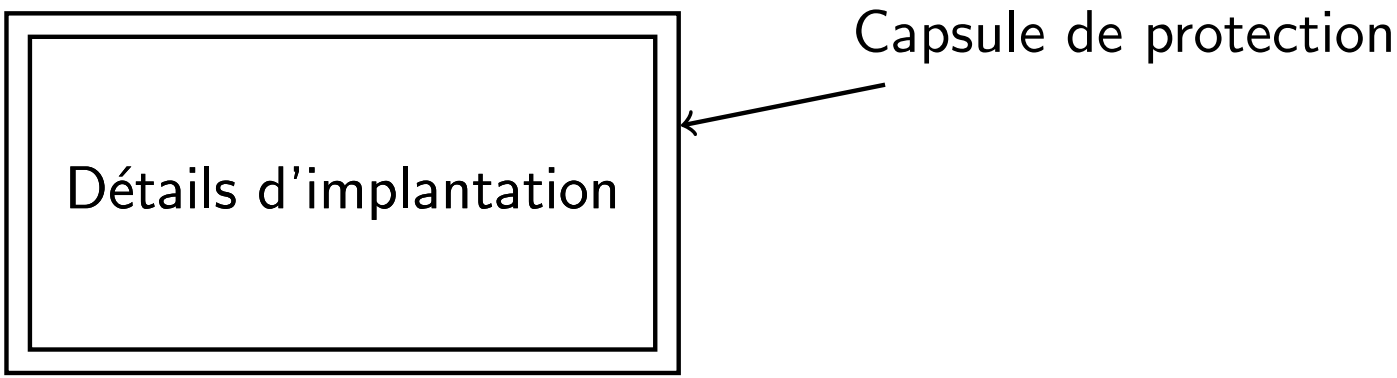
```
public class Car {
  private final String color;
  private final int seats;
  private final boolean fancy;
  ... // obvious constructor
  @Override
  public boolean equals(Object o) {
    return o instanceof Car car && fancy == car.fancy
      && seats == car.seats && color.equals(car.color);
  }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

# Encapsulation

# Encapsulation

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. On y a accès avec les méthodes accesseurs.

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

- record == classe − encapsulation

# 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.

# Exercices

# Exercice 1

Le but de ces exercices est d'écrire des classes pour représenter un magasin de vêtements de sport.
On souhaite définir un type `Clothing` qui modélise des vêtements de sport. Chaque vêtement a

- une catégorie `category` de type `String`.
- une marque (`brand`) de type `String`
- une taille `size` (un `int`)
- un prix `price` (un `int`).

Les tailles devront être comprises entre 1 et 5, et le prix doit être positif ou nul. Les champs ne seront pas modifiés.

1. Écrire un record `Clothing` avec le constructeur compact qui teste les pré-conditions.
2. Écrire un deuxième constructeur qui prend en argument la catégorie, la marque et le prix et crée le vêtement en taille 1. Pourquoi prend-on un record plutôt qu'une classe ?

# Exercice 1

Le code suivant devra fonctionner :

```
static void main() {
  var polo = new Clothing("polo", "Colmar", 3, 40);
  IO.println(polo);
  var polo2 = new Clothing("polo", "Colmar", 40);
  IO.println(polo2);
}
```

Sortie attendue :

```
Clothing[category=polo, brand=Colmar, size=3, price=40]
Clothing[category=polo, brand=Colmar, size=1, price=40]
```

# Exercice 2

Écrire une classe `SportsShop` qui modélise un magasin de sport.

Un magasin aura un champ `name` de type `String` et il contiendra une liste d'articles qui seront des vêtements de sport. Les articles pourront figurer plusieurs fois dans la liste.

1. Écrire une méthode `add` pour ajouter un vêtement dans le magasin. On pourra ajouter plusieurs fois un même vêtement dans la liste.
2. Pourquoi prend-on une classe plutôt qu'un record pour `SportsShop` ?

# Exercice 3

1. Écrire une méthode `toString` dans `SportsShop` qui permet d'afficher le magasin. On affichera le nom du magasin sur une ligne puis chaque article sur une ligne. Il ne devra pas y avoir de passage à la ligne à la fin. On devra utiliser un `StringBuilder`.

# Exercice 3

Le code suivant devra donc fonctionner :

```
static void main() {
  var polo = new Clothing("polo", "Colmar", 2, 40);
  var shirt1 = new Clothing("tshirt", "Burton", 4, 50);
  var shirt2 = new Clothing("tshirt", "Burton", 4, 50);
  var shop1 = new SportsShop("Italie2");
  shop1.add(polo);
  shop1.add(shirt1);
  shop1.add(shirt2);
  IO.println(shop1);
}
```

Sortie attendue :

```
Italie2
Clothing[category=polo, brand=Colmar, size=2, price=40]
Clothing[category=tshirt, brand=Burton, size=4, price=50]
Clothing[category=tshirt, brand=Burton, size=4, price=50]
```

# Exercice 4

1. Écrire une méthode `public int totalPrice()` qui calcule le prix total de tous les vêtements du magasin. La méthode devra renvoyer 0 si le magasin est vide.

```
IO.println(shop1.price());
// 140
```

# Exercice 5

1. Un magasin souhaite proposer des vêtements en solde. Pour cela, on écrira une méthode `onSale` qui renvoie une liste non modifiable des articles du magasin qui vont être soldés.
   Les articles soldés seront les vêtements dont la taille est supérieure ou égale à 3.

```
IO.println(shop1.onSale());
// [[Clothing[brand=Burton, size=4, price=50],
//  Clothing[brand=Burton, size=4, price=50]]
```

# Exercice 6

1. Écrire une méthode `isIncluded(SportsShop shop1, SportsShop shop2)` qui teste si tous les articles du magasin `shop1` sont aussi des articles du magasin `shop2`. Quelle est la particularité de cette méthode ?
2. Écrire une méthode `sameItems(SportsShop shop1, SportsShop shop2)` qui teste si les magasins `shop1` et `shop2` contiennent les mêmes articles, sans tenir compte des éventuelles répétitions et de l'ordre dans les listes.

```
var shop2 = new SportsShop("Jaude");
shop2.add(shirt2);
shop2.add(polo);
IO.println(SportsShop.sameItems(shop1, shop2));
// true
```