Programmation orientée objet. Cours 3

Marie-Pierre Béal et Carine Pivoteau
UGE BUT 1

Classes, encapsulation et visibilité

Mutable ou non mutable ?

```java
public record Color(int red, int green, int blue) {
  @Override
  public String toString() {
    return red + ";" + green + ";" + blue;
  }
}
```

```java
public record Palette(String name, ArrayList<Color> colors) { }
```

```java
public class Test {
  public static void main(String[] args) {
    var colors = new ArrayList<Color>();
    colors.add(new Color(0, 255, 0));
    colors.add(new Color(0, 255, 255));
    colors.add(new Color(0, 0, 255));
    var palette = new Palette("bleu-vert", colors);
    System.out.println(palette);
    // Palette[name=bleu-vert, colors=[0;255;0, 0;255;255, 0;0;255]]
   }
}
```

Mutable !

On ne peut **pas modifier les champs d'un record**, mais le contenu du record peut changer si ses champs sont eux mêmes modifiables.

```
public record Color(int red, int green, int blue) { ... }
```

```
public record Palette(String name, ArrayList<Color> colors) { }
```

```
public class Test {
  public static void main(String[] args) {
    var colors = new ArrayList<Color>();
    colors.add(new Color(0, 255, 0));
    colors.add(new Color(0, 255, 255));
    colors.add(new Color(0, 0, 255));
    var palette = new Palette("BG", colors);
    System.out.println(palette);
    // Palette[name=BG, colors=[0;255;0, 0;255;255, 0;0;255]]
    colors.add(new Color(255, 0, 0));
    System.out.println(palette);
    // Palette[name=BG, colors=[0;255;0, 0;255;255, 0;0;255, 255;0;0]]
  }
}
```

Et les listes non modifiables ?

C'est pareil. On ne peut pas modifier les références vers les objets contenus dans une liste non modifiable... mais si ces objets sont eux-même mutables, alors la liste l'est aussi.

```
import java.util.Date;
import java.util.List;

public class Test {
  public static void main(String[] args) {
    var date = new Date();
    var list = List.of(date);
    System.out.println(list);
    // [Wed Jan 31 20:57:48 CET 2024]
    date.setTime(0); // Date est mutable
    System.out.println(list);
    // [Thu Jan 01 01:00:00 CET 1970]
  }
}
```

Et si on veut des objets modifiables ?

Par exemple, on souhaite pouvoir translater un Point :

```
public record Point(int x, int y) {
  public void move(int dx, int dy) {
    x = x + dx;
    y = y + dy;
  }
}
```

Impossible : on ne peut pas modifier les champs d'un record.

Une solution est de créer un nouveau Point et de le renvoyer :

```
public record Point(int x, int y) {
  public Point move(int dx, int dy) {
    return new Point(x + dx, y + dy);
  }
}
```

Mais on peut aussi vouloir des objets "vraiment" modifiables, comme un compteur, une `ArrayList` ou une `Date`...

Les classes

Une classe est composée de

- déclarations de champs
- définitions de constructeurs
- définitions de méthodes
- déclarations d'autres classes ou records : *classes internes (nested)*.

Chaque classe est écrite dans un fichier `.java` séparé qui s'appelle obligatoirement comme la classe.

Contrairement aux records, tous ces éléments doivent être écrits explicitement dans le code de la classe.

Le record Point et la classe Point

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

```
public class Point {
  private final int x; // champ x
  private final int y; // champ y

  public Point(int x, int y) { // constructeur canonique
    this.x = x;
    this.y = y;
  }

  public int x() { // accesseur du champ x
    return x;
  }

  public int y() { // accesseur du champ x
    return y;
  }
  ...
```

La classe Point – suite

```
    ...

    @Override
    public boolean equals(Object o) {
      // explications dans les cours suivants
      return o instanceof Point other
          && other.x == x && other.y == y;
    }

    @Override
    public int hashCode() { // voir cours suivants
      return Objects.hash(x, y);
    }

    @Override
    public String toString() {
      return "Point[x=%s, y=%d]".formatted(x, y);
    }
}
```

Visibilité

En Java on utilise des modificateurs de visibilité pour les champs et les méthodes. Du moins restrictif au plus restrictif, on a :

- `public` : les champs/méthodes `public` sont accessibles partout.
- `protected` : les champs/méthodes `protected` sont accessibles seulement dans la classe elle-même et dans les classes dérivées. Ne jamais l'utiliser.
- visibilité par défaut (sans mot clé) : les champs /méthodes sont visibles dans toutes les classes du paquetage (package).
- `private` : les champs/méthodes `private` sont accessibles seulement dans la classe elle-même.

Points modifiables

```
public class MovablePoint {
  private int x;
  private int y;

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

  public void move(int dx, int dy) { // ici, aucun problème
    x = x + dx;
    y = y + dy;
  }

  @Override
  public String toString() {
    return "(" + x + "," + y + ")";
  }
}
```

Exemple d'utilisation

```
public class Test {
  public static void main(String[] args) {
    var point = new MovablePoint(1, 5);
    System.out.println(point); // (1,5)
    point.move(2, 2);
    System.out.println(point); // (3,7)
    point.move(-20, 0);
    System.out.println(point); // (-17,7)
  }
}
```

Accès aux champs

```
public class MovablePoint {
  private int x;
  private int y;
  ...
  public int x(){ return x; }
  public int y(){ return y; }
}
```

```
public class Test {
  public static void main(String[] args) {
    var p1 = new MovablePoint(2,3);
    System.out.println(p1.x()); // affiche 2
    System.out.println(p1.x);
    // ce code ne compile pas car le champ x est private
  }
}
```

Pourquoi déclare-t-on les champs `private`, ce qui nous oblige à écrire des accesseurs, plutôt que de laisser les champs `public` ?

private vs. public

```
public class MovablePoint {
  private int x;
  private int y;
  public int nbMoves; // compte le nombre de déplacements du point
  ...
  public void movePoint(int dx, int dy){
    x += dx;
    y += dy;
    nbMoves += 1;
  }
  public int nbMoves(){
    return nbMoves;
  }
}
```

```
    ...
    var p1 = new MovablePoint(2,3);
    p1.nbMoves = 1000;                    // accès au champ public
    System.out.println(p1.nbMoves()); // valeur incorrecte
```

Pas de constructeur ?

```
public class Point {
  private final int x;
  private final int y;
}
```

Il existe un constructeur par défaut

```
public class Test {
  public static void main(String[] args) {
    var p1 = new Point(); // le point (0,0) est créé
  }
}
```

Les types primitifs ont des valeurs par défaut (0, `false`, ...) et les objets sont initialisés à `null`.

Trop de constructeurs ?

On peut créer plusieurs constructeurs ou méthodes de même nom à condition que le nombre et/ou les types des arguments soient différents.

```
public class Point {
  private final int x;
  private final int y;

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

  public Point(int x){
    this(x,0);
  }

  public Point(){
    this(0,0);
  }
}
```

Plusieurs constructeurs

```
public class Test {
   public static void main(String[] args) {
     var p1 = new Point();
     var p2 = new Point(2,3);
     var p3 = new Point(2);
   }
}
```

Le mécanisme utilisé ici s'appelle la surcharge. C'est également valable pour les méthodes :

Plusieurs méthodes (ou constructeurs) peuvent porter le même nom à condition que le nombre et/ou le type des paramètres soient différents. Le type de retour n'a pas d'importance.

Remarque : la surcharge est à utiliser avec modération.

Une classe pour les segments modifiables

On veut créer une classe `MovableSegment` pour manipuler des segments déplaçables :

```
public class MovableSegment {
  private final MovablePoint start;
  private final MovablePoint end;

  public MovableSegment(MovablePoint start, MovablePoint end){
    this.start = Objects.requireNonNull(start);
    this.end = Objects.requireNonNull(end);
  }

  public void move(int dx, int dy) { // ici, aucun problème
    start.move(dx, dy); // délégation
    end.move(dx, dy);   // délégation
  }

  ...
}
```

Mot-clé `final` et classes non mutables.

On peut rendre un champ non modifiable en utilisant le mot-clé `final`. Cela signifie que l'on ne peut faire qu'une seule affectation de ce champ (lors de la construction de l'objet).

On peut aller beaucoup plus loin et fabriquer des classes non mutables (qui ne peuvent pas changer d'état, comme les records). Pour cela :

- pas de méthode permettant de modifier la valeur d'un champ,
- par sécurité, on déclare les champs `final`.
- il faut que les champs eux-mêmes soient non mutables.

Depuis Java 14, on peut obtenir le même résultat en définissant un record plutôt qu'une classe.

En Java, les chaînes de caractères sont des objets non-mutables, elles ne peuvent pas être modifiées (comme en Python). Par contre, les tableaux, les listes, les ensembles et les dictionnaires sont mutables (sauf mention du contraire dans la documentation).

Mutable ou pas ?

```java
public class Person {
  private final String firstName;
  private final String lastName;
  private int age; // attention, sa valeur peut changer

  public String firstName() {
    return firstName;
  }

  public void badMethod() {
    firstName = "toto"; // erreur à  la compilation
  }

  public void birthday() {
    age += 1; // Person est donc un objet mutable
   }
}
```

Non mutable

Penser à utiliser le mot clé `final` et ne pas mettre de “setter”.

```
public class Person {
  private final String firstName;
  private final String lastName;
  private final int age;

  public Person(String firstName, String lastName, int age){
    Objects.requireNonNull(firstName);
    Objects.requireNonNull(lastName);
    if (age < 0) {
      throw new IllegalArgumentException();
    }
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
  }

  public String firstName() {
    return firstName;
  }
}
```

Champs `private final`

Quand on écrit une classe, on doit toujours mettre

- les champs `private final`
- si vraiment un champ doit être mutable, on enlève le final.
- si on doit avoir accès au champs en dehors de la classe, on écrit une méthode accesseur public.

Encapsulation

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

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.

Quand on écrit une classe, on n'écrit pas les accesseurs pour avoir l'encapsulation, sauf si on on a vraiment besoin.

Record ou classe : comment choisir ?

- Si on a un champ non final, on doit prendre une classe
- Si tous les champs sont final, on peut hésiter entre record et classe
  - Si on ne souhaite pas d'encapsulation (la présence des accesseurs n'est pas gênante), on prend un record car beaucoup de méthodes sont déjà écrites. Par exemple `Point`, `Bird`, `Fish`, `.....`
  - Si on souhaite l'encapsulation (par exemple, un champ est une liste mutable et on ne veut pas d'accesseur à cette liste), on prend une classe (champs private final et on n'écrit pas l'accesseur à la liste).

Retour sur le mot-clé `static`

En java, le mot-clé `static` signifie que ce à quoi il s'applique n'est pas rattaché à l'objet courant (this), mais à la classe elle-même.

Les champs d'une classe peuvent être liés à la classe (`static`) ou propres aux objets qu'elle définit (pas de mot-clé).

Un champs statique est partagé par tous les objets de la classe. Il n'en existe qu'un par classe. Ça peut être, par exemple :

- un compteur du nombre d'objets d'une classe ;
- un élément particulier de la classe, par exemple une origine ;
- une constante (en utilisant `final`).

```
public class Point {
   private final int x, y;
   private static int numberOfPoints = 0;
   private static Point origin = new Point(0,0);
   private static final int MAX_NUMBER_OF_POINTS = 600;
}
```