Programmation avancée et répartie en Java : rappels sur les mots-clés de Java

Frédéric Gava

L.A.C.L
Laboratoire d'**A**lgorithmique, **C**omplexité et **L**ogique

Cours de L1 EPISEN

Plan

Plan

Plan

Déroulement du cours

Déroulement du cours

Déroulement du cours

Pourquoi du pattern-matching

Aussi présent dans de nombreux langages (OCaml, Haskell, Rust, Scala, *etc.*)

Rôles

Conditionnelle étendue (switch) pour des types de données différents. Conçu pour :

- Structures de données complexes (types algébriques, voir GADT) comme les arbres, *etc.*
- Structures de données "modulables" (comportement différent suivant les types ou les tailles de données)

Avec record et interface

Le fonctionnement est fortement lié aux record et **interface** sealed (scellées) que nous allons aussi voir.

Rappel record (1)

Aussi présent dans de nombreux langages (Ada, C, OCaml, Haskell, Rust, Scala, *etc.*)

Rôles

- En général : plusieurs données (champs donc nommés à la différence des nuplets) dans un même type
- En Java : classes simples et purement fonctionnelles/immutables (sans effet de bord, donc **final** !)

Exemples

record Animal(String nom, **int** age) { ... }

Donnera par le compilateur (intermediare)

public final class Animal **extends** java.lang.Record {

Rappel record (2)

Constructor et méthodes

```
record Animal(String nom, int age) {
        public Animal() {
                this("", 0 );
        }
        public Animal(String name) {
                this.name = name;
                this.age = 0;
        }
        // Autre codes !
        // Mais equal, hashCode, toString, assesseurs inutiles !
}
```

Immutable !

```
Animal lulu = new Animal("Lulu", 10);
int a = lulu.age(); // OUI !
lulu.age=11; // NON !
lulu.setAge(11); // NON !
record AnimalBis(couleur Pelage) extends Animal; // NON !
```

Rappel record (3)

Autres exemples

```
record MinMax(int min, int max) {};
public static MinMax(int[] data) {...}

record Point(double x, double y) {};
var p = new Point(1,1);

record Carre(Point center, double size, double angle)
 implements FigureGeometrique {
  double surface(){...}
  double perimetre() {...}
  ...
};
```

Remarque, les champs additionels d'un record doivent être statique.

Rappel record (4)

Exemple avec streams

```
// On augmente l'age de tout les animaux
        zoo = zoo.stream()
                    .map(bestiol -> new Animal(bestiol.age+1,bestio
                        .collect(Collectors.toList());
// Que les noms
        List<Strin> noms = zoo.stream()
                                    .map(Animal::nom)
                                    .collect(Collectors.toList());
// .filter(s -> ...)
// .sum()
```

Exemples simples (1)

Definition des structures

```
interface Shape { }
record Rectangle(double length, double width) implements Shape { }
record Circle(double radius) implements Shape { }
```

Definition des structures

```
public static double getPerimeter(Shape shape) throws IllegalArgumentException {
    return switch (shape) {
        case Rectangle r -> 2 * r.length() + 2 * r.width();
        case Circle c -> 2 * c.radius() * Math.PI;
        default -> throw new IllegalArgumentException("Unrecognized shape");
    };
}
```

Remarque : plus besoin de **instanceof** !

Exemples simples (2)

Choix des objets et dominance

```
static void typeTester(Object obj) {
    switch (obj) {
        case null -> System.out.println("null");
        case Integer i -> System.out.println("A integer");
        case CharSequence cs -> System.out.println("A sequence of length " + cs.length()
        case String s -> System.out.println("String");
        default -> System.out.println("Something else");
    }
}
```

Attention, le cas CharSequence domine String car chaque String est aussi une sequence (mais pas l'inverse). String est un sous-type de CharSequence. Le compilateur vérifie les dominances

Exemple plus difficile

Différencier suivant la taille

Imaginons une pile qui serait soit un tableau (si taille¡10) soit une liste.

```
static int size(Stack s) {
    switch (s.data) {
        case int[] a -> return a.length();
        case List l -> return l.length();
        default -> throw Error();
    }
}
```

Ainsi rajouter/supprimer un element pourrait transformer une pile d'un tableau en une liste (et vice-versa) suivant les besoins (un tableau sera plus efficace qu'une liste mais limité en taille ou avec un risque de cellules vides)

Default obligatoire ?

Oui...mais non, si nous avons le cas Object ou si nous avons des classes scellées (sealed **class**).

Sealed class

Rôles

Restreindre à une classe/interface quelles autres classes/interfaces pour peuvent hériter/implémenter. On définit donc bien une branche dans la hiérarchie des classes.

Intérêt

Définir une branche dans la hiérarchie des classes. On a alors bien un ensemble restreint de classes, scellées entre-elles. On peut donc traîter spécifiquement cette branche avec le switch.

Default et sealed

Plus besoin de **default** quand on sait qu'une classe est scéllée (limitée) :

```
sealed interface S permits A, B, C { }

final class A implements S { }
final class B implements S { }
record C(int i) implements S { }

static int testSealedCoverage(S s) {
        return switch (s) {
                case A a -> 1;
                case B b -> 2;
                case C c -> 3;
        };
}
```

s ne peut plus être autre chose que soit un A, un B ou un C. Il faut des classes **final** (ou record) pour maintenir la restriction de hiérarchie (branche) ou non−selead (mais cela limitera le switch).

Exemples

Des figures

```
sealed class Shape permits Circle, Square, Rectangle {}

final class Circle implements Shape { ... }
non−sealed class Square extends Shape { ... }
sealed class Rectangle extends Shape permits FilledRectangle { ... }
```

Ici, il est possible d'étendre à loisir les Carrés (danger !). Il est aussi possible d'étendre un rectangle en un rectangle plein (mais pas autrement !).

Tout un fichier

```
// Début fichier java
public sealed class Figure {}
// Toutes les autres classes seront sealed a Figure (bof)
```

Interface et Expressions

Definition d'une mini-calculatrice

On souhaite pouvoir évaluer des expressions comme $1 + (2 * 3)$:

```
sealed interface Expr permits ConstantExpr, PlusExpr, TimesExpr {
    public int eval();
}
```

Evaluation

```
final class ConstantExpr implements Expr {
    int i;
    ConstantExpr(int i) { this.i = i; }
    public int eval() { return i; }
}
```

Pattern et condition

```
static void test(Object obj) {
        switch (obj) {
            case String s && (s.length() == 1) -> ...
            case String s -> ...
            default -> ...
        }
    }
```

Pattern, sealed et services

Enumeration

```
public enum StatusEnum { SUCCESS, ERROR; }

switch (status) {
  case SUCCESS -> ...
  case ERROR -> ...
}
```

Sealed

```
sealed abstract class AbstractStatus permits ErrorStatus, SuccessStatus { }

switch (status) {
  case SuccessStatus s -> ...
  case ErrorStatus e -> ...
}
```

Au travail !