Pattern Matching

Rémi Forax

Pattern Matching

Considérée comme une technique fonctionnelle car historiquement implantée par des langages fonctionnels SML, OCaml ou Erlang

- extraire les valeurs de tuples/product types ?

Marche avec n'importe quels langages qui a un concept qui ressemble à des tuples

- class et record pour Java

Définition du Pattern Matching

Le pattern matching permet en une opération de

- tester le type/forme/valeur de la donnée (*pattern*)
- Extraire plusieurs valeurs en utilisant le *pattern* comme un masque
- Executer un code specifique avec ces valeurs

case *pattern*(*var* v1, *var* v2) -> *code*

Pattern Matching en Java

- Réhabilitation du switch
- Switch sur tout et n’importe quoi
- Patterns et Guards
- Exhaustivité
- Le switch expression

Switch en Java 1.0

Le *switch* de Java est un copier/coller du *switch* en C

switch sur les types représentés par un int (int, byte, short, char)

mais le switch en C est alien au C

```
switch(value) {
  case ‘a’:
  case ‘b’:
    ...
    break          <- fin des instructions ??
  default:
    ...
}
```

En C, si plusieurs instructions, on utilise un block entre “{“ et “}”, mais pas le *switch*, on a plusieurs instructions sans block et on utilise *break* pour indiquer la fin d’un “block”

Problèmes du switch

Ne pas avoir de block est problèmatique

- fallthrough quand on oublie un *break*
- Le scope des déclarations est tous le *switch*

```
switch(value) {
  case ‘a’:
  case ‘b’:
    var foo = 3;
    break;

  case ‘c’:
    var foo = 5;   // compile pas, il y a déjà un ‘foo’
                   // défini dans le block
    break;
}
```

Réhabilitation du switch

Le switch flèche, *switch arrow* (Java 14)

- Une flèche si une seul instruction
- Une flèche + un block si plusieurs instructions

et les *cases* peuvent être séparés par des virgules

```
switch(value) {
  case ‘a’, ‘b’ -> System.out.println("hello !");
  case ‘c’ -> {
    var foo = …
    System.out.println(foo);
  }
}
```

Le *switch* du C (le *switch colon*) existe toujours par compatibilité

On ne peut pas mélanger les case: et les case ->

Switch sur n’importe quoi

Les types possibles pour le *switch*

- byte, short, char, int (Java 1.0)
  - *switch* sur l’entier
- enum (Java 5)
  - *switch* sur ordinal()
- string (Java 7)
  - *switch* sur hashCode() + equals() si collision
- types (Java 17)
  - une cascade de *if* / *instanceof* avec un *else* à la fin

Switch et null

Par défaut, *switch* ne permet pas *null*

On peut ajouter un *case null* pour indiquer que l'on veut accepter *null*

case null ->

L'ordre des *cases* ne tient pas compte de *null* et il y a au maximum 1 *case null* par *switch*

case null, "foo" ->

défault accepte pas *null* mais on a une syntaxe spéciale

case null, **default** ->

Switch sur les String

Le *switch* sur les Strings est plus efficace que des “if equals”

```
String kind = …
Vehicle vehicle;    // doit être initialisée dans toutes les branches
switch(kind) {
  case "car" -> vehicle = new Car();
  case "bus" -> vehicle = new Bus();
  default -> throw new AssertionError();
}
```

Attention, le *default* est important car sinon la variable “vehicle” peut ne pas être initialisée (donc compile pas)

Switch sur un Enum

Le *switch* est plus efficace que des “if ==”

```
enum Color { RED, GREEN, BLUE }

Color color = …
switch(color) {
  case RED -> System.out.println("warning");
  case GREEN, BLUE -> System.out.println("ok");
}
```

Le compilateur infère que RED est Color.RED

Le switch doit être exhaustif, le compilateur

- Émet un warning si on oublie des valeurs
- Ajoute un *default* qui lève une exception si toutes les valeurs sont listées

Switch sur des objets

On peut faire un *switch* sur n’importe quel type

```
Object json = …
switch(json) {
  case JSONObject object - > ...
  case JSONArray array - > …
  default - > ...
}
```

Le switch doit être exhaustif sinon il y a une erreur

Equivalent à if … instanceof

Un switch sur des objets est équivalent à
une cascade de *if* … *instanceof*

```
Object json = …
switch(json) {
  case JSONObject object - > ...
  case JSONArray array - > …
  default - > ...
}
```

est équivalent à

```
if (json instanceof JSONObject object) {
  ...
} else
  if (json instanceof JSONArray array) {
    ...
  } else {   < – pas de instanceof pour la dernière branche
    // default
  }
```

Patterns

instanceof et *switch* utilisent les patterns suivants

- Type Pattern
  - Type variable
  - var variable (Java 18)
- Guard Pattern
  - *pattern && condition*
- Parenthesis Pattern
  - (*pattern*)
- et pour Java 18: Record Pattern et Array Pattern
  - Type(var x, var y) // record avec 2 components
  - Type[] { var x, var y } // tableau à 2 valeurs
  - Type{} { var x, …} // tableau à au moins 1 valeur

Guards

On peut vouloir compléter un *case* avec un *guard*

Par exemple si on a plusieurs sorte de cookies

```
record Cookie(boolean chunky) implements Cake { }
```

On peut vouloir sélectionner seulement certain cookies

```
Cake cake = ...
switch(present) {
  case Cookie cookie && cookie.chunky -> …
  case Cookie cookie -> …
   ...
}
```

Guards et variables locales

Comme avec les lambdas, une variable utilisée dans un *guard* doit être effectivement final (assigné une seul fois)

L'exemple suivant ne compile pas

```
for(var index = 0; index < array.length; index++) {
  switch(o) {
    case String s && index == 3 -> …
    default -> ...
  }
}
```

"index" ne peut pas changer de valeur et être utilisé dans un guard

Guards et ordre

Un *case* avec un *guard* doit être placé avant le même *case* sans *guard*

Le code suivant compile pas

```
Cake cake = ...
switch(present) {
  case Cookie cookie -> …
  case Cookie cookie && cookie.chuncky -> …
   // doit être placé AVANT
  ...
}
```

Les *cases* avec des *guards* différents n’ont pas d’ordre

Exhaustivité (*completeness*)

Le compilateur demande à ce que les switchs soit exhaustif

Détermine si tous les cas couvrent toutes les valeurs possibles sinon plante (ou warning)

- *default* veut dire toute les valeurs sauf *null*
- un *total pattern* veut dire toutes les valeurs avec *null*
- Si il y a un nombre fini de valeur/types possible
  - Les constantes d'un *enum*
  - Les types "*permits*" d'un *sealed* types

  alors le compilateur insère un *default* qui plante

Total Pattern

Un total pattern est un *case* qui couvre toutes les valeurs possible du switch

```
Number number = ...
switch(number) {
  case Integer i -> ...
  case Double d -> …
  case Number n -> … // total pattern,
                     // le switch devient exhaustif
}
```

Un *total pattern* accepte *null*, donc le *switch* ci-dessus accepte *null*

Case var vs *case total pattern*

On préfère utiliser la syntaxe **case var** au lieu de *case total pattern* pour marquer que le switch est nullable

```
Number number = ...
switch(number) {
  case Integer i -> ...
  case Double d -> …
  case var number -> … // le switch devient exhaustif
}
```

même si *case var* et *case* Number ont le même sens

Le *case var* devrait ajouté en Java 18 !

Switch et ordre des *cases*

Les *cases* doivent être du plus précis au moins précis

un *case* ne doit pas “dominer” un *case* précédent

Par ex, le *switch* ci-dessous ne compile pas

```
Object o = …
switch(o) {
  case CharSequence seq - > ...
  case String s - > ...
  default - > ...
}
```

car String est plus précis que CharSequence

Exemple avec un sealed type

En introduisant une interface sealed entre les types, le switch devient exhautif sans *default* ni *total pattern*

```
sealed interface Shape permits Circle, Rectangle { }
record Circle(int radius) implements Shape { }
record Rectangle(int width, int height) implements Shape { }

double surface(Shape shape) {
  switch(shape) {
    case Circle c -> { return Math.PI * c.radius * c.radius; }
    case Rectangle r -> { return r.width * r.height; }
  }
}
```

On écrit pas de *default* ou de *total pattern*, comme cela, le compilateur plante si on ajoute un nouveau sous-type

Le switch expression

Switch expression

On peut aussi utiliser le *switch* comme une expression

Au lieu de

```
Vehicle vehicle;
switch(kind) {
  case "car" -> vehicle = new Car();
  case "bus" -> vehicle = new Bus();
  default -> throw new AssertionError();
}
```

On peut écrire

```
var vehicle = switch(kind) {
  case "car" -> new Car();
  case "bus" -> new Bus();
  default -> throw new AssertionError();
};    //  il y a un point virgule là !
```

yield

Le *switch* expression permet aussi la syntaxe *case:* dans ce cas, on a besoin de renvoyer la valeur

```
var vehicle = switch(kind) {
  case "car": yield new Car();
  case "bus": yield new Bus();
  default:  throw new AssertionError();
};
```

yield permet d'indiquer la valeur retournée par le switch

on peut pas utiliser **return** qui renvoie la valeur de la méthode

Block flèche et Yield

Pour un *switch* expression avec des flèches, si on veut renvoyer la valeur d'un block, on utilise aussi yield

```
var vehicle = switch(kind) {
  case "car" -> {
    System.out.println("create a car !");
    yield new Car();
  }
  case "bus" -> new Bus();
  default:  throw new AssertionError();
};
```

Relation avec le polymorphisme

Pattern Matching et Polymorphisme

Le polymorphisme et le pattern matching sont dual

```
sealed interface Pet {
  long price();
  record Cat(String name)
    implements Pet {
    long price() { return 100; }
  }
  record Dog(String name)
    implements Pet {
    long price() { return 200; }
  }
}
```

```
sealed interface Pet {
  record Cat(String name)
     implements Pet {}
   record Dog(String name)
      implements Pet {}
}
...
static long price(Pet pet) {
  return switch(pet) {
    case Cat cat -> 100;
    case Dog dog -> 200;
  };
}
```

Polymorphisme vs Pattern Matching

Si on veut une hierarchie

- ouverte (pas *sealed*)

  on veut laisser la possibilité d'**ajouter** des nouveaux **sous-types**

  On utilise le polymorphisme

- fermée (*sealed*)

  on laisse la possibilité d'**ajouter** de nouvelles **opérations**

  On utilise le pattern matching

cf the expression problem

https://en.wikipedia.org/wiki/Expression_problem

Et dans le future proche
(après Java 17)

Switch objets et constantes

Dans le futur, on pourra mixer des constantes avec le *switch* sur des objets

```
String sayHello(String name) {
  switch(name) {
    case "Bob" -> { return "Hi Bob !"; }
    case String s -> { return "Hello " + name + " !"; }
  }
}
```

Destructuration des records

Ajout d’un pattern supplémentaire pour extraire les valeurs d’un record une fois celui-ci reconnu

Avec un record

```
record Cookie(boolean chunky, long price)
```

On pourra écrire

```
Cake cake = ...
switch(present) {
  case Cookie(var delicious, var price) -> {
    System.out.println("delicious: " + delicious);
  }
  ...
}
```

Le compilateur va demander d’extraire la valeur “chunky “du cookie en appelant le l’accesseur correspondant à la propriété

Destructuration lors de l’assignation

On peut étendre le matching de *records* à l’assignation

Avec un record

record Cookie(boolean chunky, long price)

On pourra écrire

```
Cookie cookie = …
Cookie(var delicious, var price) = cookie;
System.out.println(delicious + " " + price);
```

qui permet d’extraire les valeurs sans appeler explicitement l’accesseur du *record*