Java 17, le record & pattern matching c’est pour maintenant ?

Table of Contents

Promis juré, cet article n’est pas une liste des "nouvelles fonctionnalités de Java" (JLS[1] / JSR[2] / JEP[3]).

Depuis sa sortie en septembre 2021, les articles sur Java **17** pleuvent.

Ok, ça y est, on a bien compris que cette version est une LTS[4].

Mais c’est aussi bien plus que cela.
C’est une milestone de l’objectif ambitieux nommé
"Record and Array Pattern Matching".

Cet objectif est un ensemble de fonctionnalités synergiques :

- Les `instanceof` avec "Type Patterns" (dispo en 16)
- Les Switch on Patterns (Java 17 preview, et probablement dispo en 19)
- La déconstruction de `record` (Peut-être Java 19 preview)
- La déconstruction d’array (Java 19+ ?)
- Les imbrications de patterns (pas de visibilité de dispo)

C’est donc de ces features dont on parle ici :

- À quoi elles servent ?
- Comment et dans quels contextes les utiliser ?
- Qu’apportent-elles à notre code ?

Et avec bien sûr des exemples de code !

Révolutionner Java, oui, mais pourquoi ?

Quand on souhaite améliorer un produit, on commence par se demander où les efforts seraient les plus bénéfiques.

Aujourd'hui le design de nos backend d'applications de gestions pousse (□) autour d'une problématique :
Faire varier des comportements en fonction de cas d'usage

Dans ce genre d'applications, quelle que soit l'architecture choisie ou le style dev, on se retrouve à un moment ou un autre à :

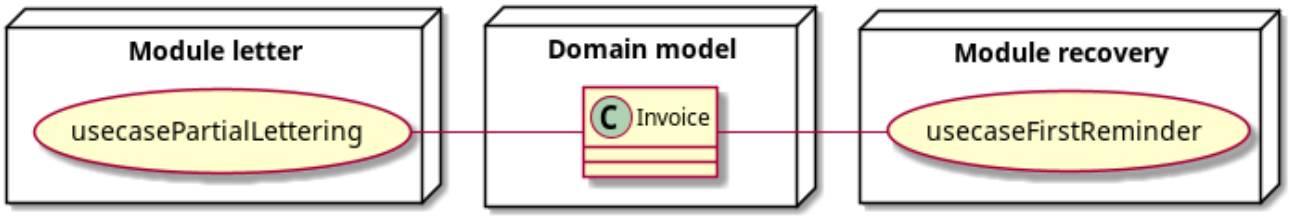
1. **Modéliser notre domaine métier**

   Cela peut être fait dans un package spécifique avec des POJO, ou avec des `Entity` JPA.
   La seconde option est la plus répandue, mais ce n'est pas ma préférée. Je trouve que c'est une erreur de concevoir le business d'une application autour d'une base de données.

2. **Écrire des DTO**

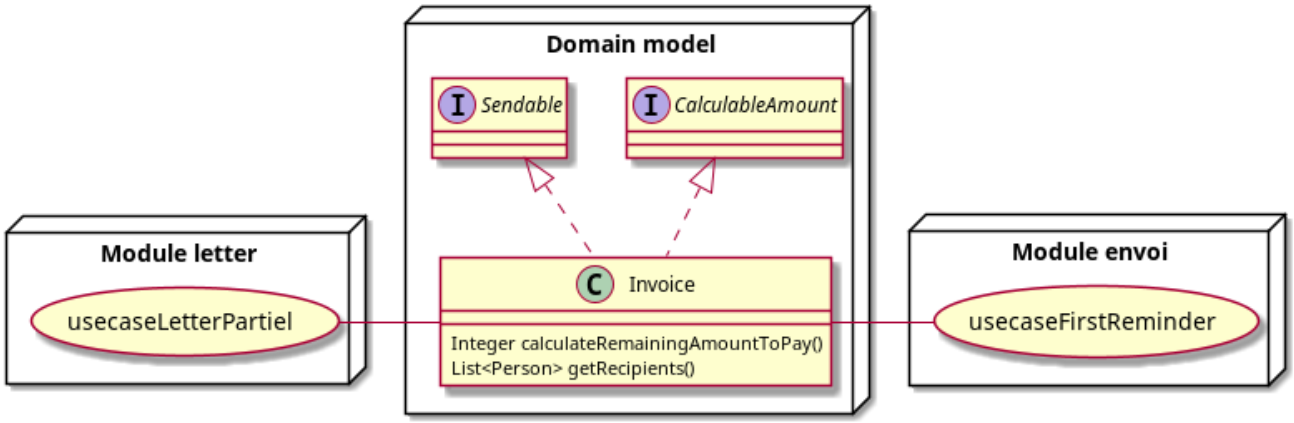
   Dans l'idéal, un dto est immutable (Il n'y a aucune raison de changer la représentation d'une donnée transmise à un moment T).
   Le `record` est la structure de données la plus appropriée.
   Sinon, avant Java 14, on a les `@Value` de Lombok.
   On peut aussi se contenter de POJO mutables.

Dans nos applications modulaires, on peut avoir envie de partager ces structures de données entre des modules :



Et bien pour faire varier les comportements des actions affectant ces classes, la POO nous incite à ajouter des méthodes sur nos classes de domaine.

Par exemple, pour la domain-class `Invoice`, le module `Letter` pourrait vouloir ajouter une méthode `calculateRemainingAmountToPay()`.
Le module envoie pourrait vouloir une méthode `getRecipients()`.




```
class Invoice implements CalculableAmount, Sendable {
  private String label;
  private CodeInvoice code;
  private Client client;
  private Receipt receipt;
  private Devise amount;

  @Override
  public Integer calculateRemainingAmountToPay(){
    ...
  }

  @Override
  public List<Person> getRecipients(){
    ...
  }

  ...
}
```

Au bout d'un moment, notre domain-class `Invoice` a beaucoup de méthodes issues de différents modules.
Le module `Letter` utilise `Invoice` et se retrouve à pouvoir appeler les méthodes du module `Send` ; ce qui viole au moins :

- Le principe de ségrégation des interfaces
- Le principe de responsabilité unique (Car la classe Document a maintenant 2 raisons de changer : le contexte `Letter` et le contexte `Send`)

Effet bonus : Quand on change `Invoice` dans le cadre du contexte `Letter`, on doit recompiler/relivrer aussi le contexte `Send`.

Solution : séparer la logique métier des structures sur lesquelles elle agit

Pour y parvenir, on utilisait jusque-là au moins ces 3 patterns :

- Le visitor pattern[5]
- Le delegate pattern
- Le pattern service-everywhere avec des méthodes à 8 arguments (un anti-pattern d'après moi), qui nait de la programmation procédurale dans un monde d'inversion de contrôle.

Mais à présent avec Java 17, une quatrième solution élégante s'offre à nous : Le Pattern Matching.

Mais qu'est-ce que le pattern matching ?

Je pense qu'on ne peut pas couper à la définition de Wikipédia :

> In computer science, pattern matching is the act of checking a given sequence of tokens for the presence of the constituents of some pattern.
>
> — https://en.wikipedia.org/wiki/Pattern_matching

On a tendance à penser alors aux expressions régulières, mais non, il ne s'agit pas de cela.

Là, les patterns à matcher sont des structures de données :

- Des classes
- Des interfaces
- Des array
- Et bien sûr des records !

Je trouve que le cas du matching sur `instanceof` avec Type-Pattern est le plus facile à comprendre. Avant Java 17, on avait ça :

```
if (invoice instanceof PaidInvoice) { // Oui je mets des accents dans mon code
français. La sémantique !
  letterService.letter(((PaidInvoice) invoice));
  return;
}
if (invoice instanceof DueInvoice) {
  recoveryService.remind(((DueInvoice) invoice));
}
```

Et à présent :

```
if (invoice instanceof PaidInvoice paidInvoice) {
  letterService.letter(paidInvoice);
  return;
}
if (invoice instanceof DueInvoice dueInvoice) {
  recoveryService.remind(dueInvoice);
}
```

Ici le pattern à matcher est l'appartenance aux classes `PaidInvoice` et `DueInvoice`. On teste si l'instance a un des types, et un cast implicite est fait vers une "binding variable" (`paidInvoice` ou `dueInvoice`).

Comment le Pattern Matching remplace-t-il le visitor pattern ?

J'ai promis des exemples de code, les voici.

Voici l'implémentation du visitor pattern avec le modèle de `Invoice` :

```
interface InvoiceVisitable {
  default void accept(InvoiceVisitor invoiceVisitor) {
    invoiceVisitor.visit(this);
  }
}

abstract class Invoice implements InvoiceVisitable {
}

class PaidInvoice extends Invoice {
}

class DueInvoice extends Invoice {
  private Integer reminderNumber = 0;

  public void incrementReminderNumber(){
    reminderNumber++;
  }

  public boolean hasAlreadyBeenReminded() {
    return reminderNumber >= 1;
  }
}

interface InvoiceVisitor {
  void visit(PaidInvoice paidInvoice);

  void visit(DueInvoice dueInvoice);
```

```
}

interface LetterService {
  void letter(PaidInvoice paidInvoice);
}

interface RecoveryService {
  void remind(DueInvoice dueInvoice);
}

record MainInvoiceVisitor(LetterService letterService, RecoveryService
recoveryService) implements InvoiceVisitor {

  @Override
  public void visit(PaidInvoice paidInvoice) {
    letterService.letter(paidInvoice);
  }

  @Override
  public void visit(DueInvoice dueInvoice) {
    recoveryService.remind(dueInvoice);
  }
}

record InvoiceService(MainInvoiceVisitor mainInvoiceVisitor) implements
InvoiceProcessing {

  public void handleInvoice(Invoice invoice) {
    invoice.accept(mainInvoiceVisitor);
  }
}
```

On observe que le rapport code utile / boilerplate n'est pas excellent.

Et maintenant :

```
record InvoiceService(LetterService letterService, RecoveryService recoveryService)
implements InvoiceProcessing {

  public void handleInvoice(Invoice invoice) {
    if (invoice instanceof PaidInvoice paidInvoice) {
      letterService.letter(paidInvoice);
      return;
    }
    if (invoice instanceof DueInvoice dueInvoice) {
      recoveryService.remind(dueInvoice);
    }
  }
}
```

Le `InvoiceService` se suffit à lui-même, et la lisibilité me semble très acceptable.

Mais avez-vous remarqué quelque chose dans ce dernier bout de code ?

Le cas où `invoice` est d'un autre type n'est pas géré !
Et non, pas de `throw new NotImplementedException()` cette fois-ci.

C'est là que la fonctionnalité Java 15 de types scellés intervient.
Modifions un peu notre modèle :

```
abstract sealed class Invoice permits PaidInvoice, DueInvoice {
}

final class PaidInvoice extends Invoice {
}

final class DueInvoice extends Invoice {
  private Integer reminderNumber = 0;

  public void incrementerNombreReminder(){
    reminderNumber++;
  }

  public boolean hasAlreadyBeenReminded() {
    return reminderNumber >= 1;
  }
}
```

Traduction en français :

> Il n'existe que 2 types de `Invoice` possibles : `PaidInvoice` et `DueInvoice`.
> Ces dernières ne peuvent être étendues.
> Point.

Cela donne donc :

```
record InvoiceService(LetterService letterService, RecoveryService recoveryService)
implements InvoiceProcessing {

  public void handleInvoice(Invoice invoice) {
    switch (invoice) {
      case PaidInvoice paidInvoice -> letterService.letter(paidInvoice);
      // case DueInvoice dueInvoice -> recoveryService.remind(dueInvoice);
    }
  }
}
```

J'ai commenté le cas de la `DueInvoice` afin d'observer ce que nous disent le compilateur et l'IDE :

On doit alors déclarer le `Consumer<? extends Invoice>` de tous les cas restants, ou bien les grouper dans un `default` :

```
record InvoiceService(LetterService letterService, RecoveryService recoveryService)
implements InvoiceProcessing {

  public void handleInvoice(Invoice invoice) {
    switch (invoice) {
      case PaidInvoice paidInvoice -> letterService.letter(paidInvoice);
      // case DueInvoice dueInvoice -> recoveryService.remind(dueInvoice);
      default -> LOGGER.info("Cool y a rien à faire pour le cas là !");
    }
  }
}
```

Avec cette syntaxe, le langage nous apporte une validation métier de plus à la compile time (soit plus tôt qu'à la runtime. Tout ce qui réduit la boucle de feedback est bénéfique).
C'est les TDDistes qui sont contents.

Et si on allait encore plus loin ?
Allez, ajoutons une feature preview de Java 17 : un "Guarded Pattern"

```
record InvoiceService(
    LetterService letterService,
    RecoveryService recoveryService)
    implements InvoiceProcessing {

  public void handleInvoice(Invoice invoice) {
    switch (invoice) {
      case PaidInvoice paidInvoice -> letterService.letter(paidInvoice);
      case DueInvoice dueInvoice && dueInvoice.hasAlreadyBeenReminded() ->
recoveryService.startRecovery(dueInvoice);
      case DueInvoice dueInvoice -> recoveryService.remind(dueInvoice);
    }
  }
}
```

Un "Guarded Pattern" permet d'ajouter à notre pattern des conditions sur les valeurs de l'objet matché en plus de son type.

Alors c'est très bien tout ça, mais l'objectif à terme du pattern matching va encore plus loin en ce qui concerne les records.

Reprenons notre exemple de `Invoice`, mais considérons qu'elle vient d'arriver d'un `Controlleur` sour forme de DTO (et donc de record) :

```
record Invoice(String code, String label, Integer amount, ZonedDateTime creationDate,
...){}
```

Je ne lui donne que quelques champs, mais considérons en plus qu'il y a en une vingtaine, une centaine, beaucoup…

Quand je veux mapper cette invoice vers un usecase, alors ce dernier n'a très certainement besoin que de seulement quelques-uns de ces champs. Le code suivant serait donc une erreur de design :

```
@RestController
class InvoiceControlleur {

  @PostMapping
  @ResponseStatus(HttpStatus.CREATED)
  public Long create(@RequestBody Invoice invoice) {
    Preconditions.checkNotNull(invoice);
    notifyNewInvoiceUseCase.handle(invoice);
    return invoiceService.handle(invoice)
  }
}
```

Après Java 18 (Java 19 avec un peu de chance), on va pouvoir déconstruire des structures de données.

Qu’est-ce que la "déconstruction"

Ce concept a un objectif similaire au I de SOLID : la ségrégation.

Si je reçois un objet avec 43 champs alors que j’en ai besoin que de 2, la "*deconstruction on pattern*" va m’aider.

Regardons ça avec du code.

J’ai mon énorme dto receipt :

```
record Invoice(
  String code,
  String libellé,
  Integer amount,
  ZonedDateTime dateCréation,
  ... // imaginez ici 39 autres champs
){}
```

Mais la règle métier que je veux appliquer ne porte que sur le `code` et le `amount`. Je peux alors étendre le concept de `instanceof` précédent, en lui ajoutant une déconstruction du Record "Invoice" :

```
if (object instanceof Invoice(String code, Integer amount)) {
  myUseCase.handle(code, amount);
}
```

Ici, `type` et `price` sont des "binding variables" générées implicitement si l’`object` match le pattern `Product`.

Et ça sert à quoi ?

1. Découplage
2. Expressivité

Je ne compte pas expliquer ici en quoi ces 2 principes logiciels sont bénéfiques.

Comparez plutôt avec la méthode habituelle :

```
if (object instanceof Invoice) {
    Invoice invoice = ((Invoice) object);
    String type = invoice.getType();
    String price = invoice.getPrice();
    myUseCase.handle(type, price);
}
```

Qu’apporte à notre code ces nouvelles fonctionnalités ?

- Plus de validation à la compile-time, et donc une boucle de feedback plus rapide.
- Développer plus intuitivement (le compilateur nous dis ce qu’on a oublié)
- Faire émerger de meilleurs designs

Dans quels langages on en trouve déjà

- Scala
- Rust

Conclusion

[1] JLS : Java Language Specification

[2] JSR : Java Specification Request

[3] JEP : JDK Enhancement Proposal

[4] LTS : Long Term Support

[5] "Today, to express ad-hoc polymorphic calculations like this we would use the cumbersome visitor pattern". source : https://openjdk.java.net/jeps/405