Value Types et Pattern Matching

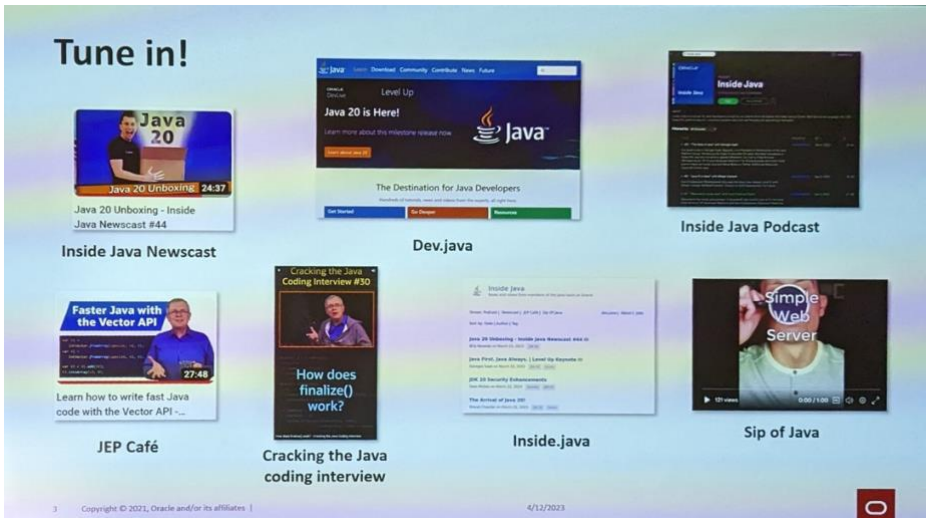
Remettre les données au centre des applications

Date : 12 avril 2023
Speakers :

- José Paumard, Java Développer Advocate Oracle
- Rémi Forax, maître de conférences à l'université Gustave Eiffel)

Format : Université de 3h

Concept remis en avant dans cette conférence : Data Oriented Programming
Publicité de José pour le site https://dev.java/ géré par l'équipe des dév rel d'Oracle. Site lancé à l'occasion de la sortie de Java 17.
Autres pubs de José pour des chaînes Youtube et des podcasts : Inside Java Newscast, JEP Café, Cracking the Java coding interview, Sip of Java, Inside Java Podcast. Et encore une : dev.java/community/ => tous les Java User Group y sont référencés, dont les 23 en France (la plus grosse communauté :-)



Dernière pub : openjdk.org : les dernières early access y sont accessibles, dont le JDK 21 encore incomplet. JavaFX est désormais distribué sur OpenJDK

Ses dernières années, l'équipe Java cherche à moderniser le langage et la plateforme Java en introduisant de la programmation fonctionnelle ou bien encore en s'adaptant au Cloud. Ces évolutions sont réparties dans plusieurs projets de l'OpenJDK :

- **Amber** : projet un peu fourre-tout incluant le Data Oriented Programming, l'interpolation de chaine de caractères …
- **Valhalla** + **Lilliput** : permet à Java de coller davantage à l'architecture des CPU modernes. Les objets ont un header qui prennent de la mémoire. Le header (en général 128 bits) de la classe java.lang.Integer est plus gros que l'int qu'on stocke à l'intérieur. Lilliput vise à diminuer cette empreinte mémoire en passant à 64 bits, voire 32 bits
- **Leyden** : création d'une image statique du JAR au moment d'installation.
- **CRaC** : idée venant de Smalltalk permettant de stocker l'état de l'application en mémoire pour gagner en temps de démarrage. Ne fonctionne que sur Linux (utilisation de Cryo).
- **Galahad** : projet qui place GraalVM dans l'OpenJDK
- **Panama** : appeler du C est historiquement difficile. Aujourd'hui on a paradoxalement moins peur du C. Foreign Function API. Le JIT génère le code de glue.
- **Vector API** : meilleure exploitation des CPU modernes. Sur des serveurs, permet de gagner en performance jusqu'à un facteur 16. Le JIT permet de profiter de cette API. Rémi rappelle que l'auto-vectorisation existe depuis au moins Java 7.

Projet Amber

Data Oriented Programming dans le contexte de Java (et pas dans SQL).
A son origine, le langage Java a été conçu de manière à avoir un C avec des objets sans être de C++. Les premiers développeurs Java sont des réfugiés de C++ et de Smalltalk.
Object Oriented Protramming (OOP) de Java basée sur les notions de : classe, encapsulation, interface, sous-typage, le polymorphisme (tardif). Le code Java passe son temps à appeler des APIs (les interfaces). Depuis 25 ans, l'OOP rencontre une belle success story. Aussi, pourquoi ne pas utiliser que de l'OOP ? En Java, les APIs sont plus importants que le code. Dans sa vie de tous les jours, un développeur utilise des API, mais en crée moins souvent (sauf ceux qui développent des librairies). Au quotidien, le développeur manipule des données. Le Data Oriented Programming se focaliser sur les données : les données sont plus importantes que le code. La modélisation des données est essentielle. Les données changent souvent, c'est naturel. Cela alors qu'une API est stable et change moins souvent. Lorsque les données changent, on a besoin que le compilateur nous aide.

Rémi et José vont illustrer leurs propos avec du code.
Ils partent de classes City et Department implémentant l'interface Named ayant une méthode name() qu'il est est nécessaire d'implémenter.

```
interface Named {
  String name() ;
}

public class City implements Named {
   private final String name;
```

```
}


public class Department implements Named {
   private final String name;
   private final List<City> cities;

public Department(String name, List<City> cities) {
    this.name = Objects.requireNonNull(name);
    this.cities = .copyOf(cities); // Contrôle les nullités

  }

}
```

Rémi sort les données de l'interface et retire la méthode name(). On se retrouve avec une méthode statique name() faisant 2 instanceof :

```
interface Named { }

static String name(Named named) {
   if (named instanceof City) {
      var city = (City) named;
      if (city.population() < 0) {
        throw new ISE("danger danger");
      }
      return city.name();
   }
   if (shape intanceof Department)
     var department = (Departement) named;
     return department.name();
   }
  throw new AssertionError();

}
```

Suit un refactoring montrant l'usage d'Amber : record, pattern matching …

1. On retire la méthode name() de l'interface Named
2. Le code métier des méthodes name() de City et Department sont déplacés dans une méthode statique name faisant des instanceof de City et Department
3. Utilisation du pattern matching
   if (named instanceof City city) { return city.name() };
   Plus besoin du cast. IntelliJ détecte ce refactoring.
   Ce n'est pas qu'un sucre syntaxique, mais une modification profonde du langage Java qui va apporter plein de nouveautés. D'après José : peut-être aussi important que les lambdas apportés par Java 8. Rémi pense que le pattern matching sera même un plus gros changements que les Lambda. Ce changement arrivera petit bout par petit bout ans les prochaines versions de Java.

4. Ajout d'une classe Region implémentant Named : le code business continue à compiler. Le compilateur aurait signalé le problème avec les interfaces.
5. Le compilateur doit connaître le nombre d'implémentation de l'interface Named. On doit sceller l'interface Named (**sealed** en anglais). Les classes scellées doivent être finales (héritage interdit). Possibilité de créer des classes abstraites non-sealed class. Autre contrainte : toutes les classes scellées doivent être dans le même module Java. Le compilateur doit voir toutes les classes.
   public sealed interface Named permits City, Department { }
   Ces vérifications sont effectuées à la compilation et au runtime. Le bytecode généré par ASM sera donc vérifié.
6. On ajoute un switch dans le code métier. Le switch est devenu une expression et supporte le pattern matching. L'ancien switch venait du C et est très peu utilisé dans le JDK.
```
return switch(named) {
   case City city -> city.name(); // City city est un pattern
   case Department department -> department.name();
}
```
   Plus besoin d'utiliser de default car le compilateur sait que le switch est complet. On peut quand même ajouter un default dans le cas où le dév ne compile pas tout d'un coup (mais est-ce une bonne pratique ?).
7. L'ajout d'une classe Region fait échouer le compilateur. Plus besoin de grep. Lorsque le modèle de données évolue, le compilateur permet de trouver où modifier le code. Bonne pratique : ne surtout pas ajouter de default avec les switchs utilisant des classes scellées, car sinon le compilateur ne prévient plus.
8. Utilisation des records pour agréger des données ensemble. Fonctionnalité « Convert to record » d'IntelliJ. L'utilisation d'un record sur la classe City permet de retirer 35 lignes de code. Dans le langage Java, on a tout un modèle derrière les records.
9. Introduction d'un **constructeur compact**
```
public record City(String name, int population) implements Named {
  public City {
       Objects.requireNonNull(name);
        if (population < 0) throw new IllegalArgumentException("Nope");
   }
}
```
   Le compilateur ajoute le *this.name = name* à la fin du constructeur. La désérialisation va appeler ce constructeur compact. Chose qui n'était pas le cas avec les classes Java. Les **data class** Lombok et Kotlin n'apportent pas la sécurité des records.
10. Refactoring en record du Department. Un record est final. Un record ne peut ni être étendu, ni étendre d'une classe. Un record étend d'une classe Record qui lui-même étend java.lang.Object. Un objet record ne peut pas avoir d'autres états
11. Dans le code métier, on peut utiliser le **record pattern** (feature en preview)
```
return switch(named) {
   case City(String name, int population) -> name; // Déconstruction
   case Department(String deptName) -> deptName ; // Le nom de la variable est libre
}
```
   Lorsque les données changent dans le record City, le code ne compilera plus. Exemple : on ajoute un tableau de City[] dans le record Department => nécessité d'ajouter un argument dans le switch :
   `case Department(String deptName, City[] cities) -> deptName;`

```
public Departement {
  Objects.requireNonNull(name) ;
  cities = cities.clone() ; // Programmation défensive
}

public City[] cities() {
    return cities.clone ; // Double défense
}
```

12. Refactoring avec le mot clé **var** : affaiblit le travail de validation offert par le compilateur. Pas une bonne idée d'utiliser le var dans le record pattern. Exemple avec l'ajout d'un record Population
    ```
    case Department(var deptName, var cities) -> deptName;
    public record Population(int amount) {
    }
    ```
13. **Syntaxe _** à venir pour les paramètres inutilisés :
    ```
    case City(var name, _) -> name;
    ```
14. Utilization du **case null** :
    ```
    return switch(named) {
       case null -> "Ha ha ha!";
    }
    ```
15. Utilisation du mot clé **when** :
    ```
    case City(var name, var population)
      when name.equals("Oudon") -> "Oudon";
    ```
    A noter que l'ordre des case est important

Le record patterns va permettre de détecter des modifications structurelles.
Les types scellés permettent d'avoir des switchs exhaustifs.

Dualité entre le polymorphisme et le pattern matching (Phill Wagner) :

- Polymorphisme : ajout de sous-types mais sans nouvelle opération
- Pattern matching : ajout de nouvelles opérations, mais pas de sous type à cause des interfaces scellées.

L'aide du compilateur diminue l'étendue des tests à faire.

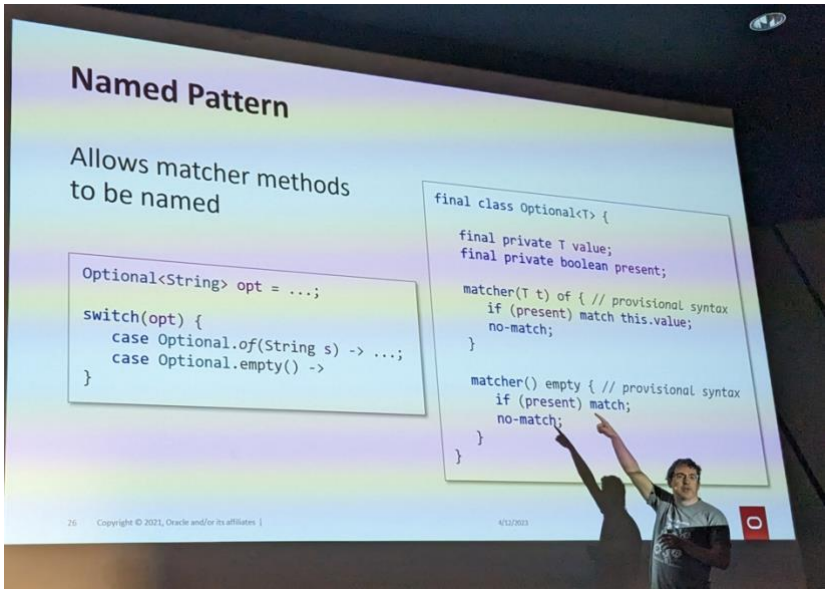
Dans le futur, il est prévu de faire des record pattern sur des classes
Déconstructeur : on transforme un Point en 2 entiers.

```
class Point {

  private int x, y ;
  matcher(int x, int y) Point {    // syntaxe provisionnelle
     matches this.x, this.y ;
  }
}
```

Exemple sur le Map.Entry sur la **déstructuration** impérative lors des assignements de variables :
Point p = … ;

let Point(int x, int y) = p ;
for (let Map.Entry(var key, var value) : entrySet) { … }



Mes notes s'arrêtent là. Mon écran de PC portable a pris l'eau pendant la pause. Impossible de le redémarrer avant le lendemain.