



# Atelier 7 – l’introspection

## Table des matières

1	Objectifs.....	2
2	Théorie .....	2
3	Introduction.....	2
4	Exercices.....	3
4.1	Attributs (Fields).....	3
4.2	Méthodes (Methods).....	4
4.3	Choix de classe “dynamique” .....	5
4.4	Création du graphe d’instances .....	5
4.5	Annotations.....	6
4.6	Analyse du graphe d’instances .....	6
4.7	 Bonus : Types génériques .....	7
4.8	 Bonus : graphe complet.....	7

## 1 Objectifs

ID	Objectifs
AJ01	Comprendre, et savoir utiliser l'API d'introspection en JAVA
AJ02	Utiliser et créer des annotations

## 2 Théorie

La théorie sur l'introspection a été vue en cours de Concepts Orienté Objet et ne sera pas détaillée ici. Cependant, n'hésitez pas à utiliser également les ressources à votre disposition. A savoir :

- La javadoc officielle :  
<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/reflect/package-summary.html>
- Les guides techniques fournis par Oracle :  
<https://www.oracle.com/technical-resources/articles/java/javareflection.html>

Pour un exercice, nous allons créer une annotation personnalisée. Pour savoir comment faire, et comprendre à quoi les annotations peuvent nous servir, nous vous conseillons de lire la théorie sur les annotations, disponible sur Moodle.

## 3 Introduction

Dans ce projet, nous allons développer une API pour permettre la description de classes JAVA en JSON. Cette API nous permettra ensuite de générer des représentations UML de nos classes, automatiquement, sur base de notre JSON.

Dans un second temps, nous allons faire de même pour des graphes d'instance.

Notez d'emblée la grosse différence entre ces deux "parties" :

- d'un côté nous analysons des classes ;
- de l'autre nous analysons des instances, ce qui nécessite de faire des "new" avant de lancer l'analyse.

Cette API sera développée à l'aide d'un serveur web "embedded" nommé Grizzly. "embedded" signifie ici que notre serveur web est intégré dans notre application JAVA. On peut donc le lancer, comme n'importe quel projet, en appuyant sur le bouton "run" de votre IDE favori. À l'inverse, les "application server" sont des serveurs à installer sur votre machine et auxquels on donne notre projet pour qu'il soit exécuté. Cette seconde option est plus lourde et complexe à mettre en place. C'est pourquoi nous avons opté pour la première option ici.

Ce serveur web, ainsi que tout ce qui tourne autour, sera intégré dans notre projet à l'aide d'une dépendance MAVEN.

Ne vous inquiétez pas ! Toute cette partie “web” vous est fournie dans le projet. Vous allez vous occuper uniquement de la partie “business”.

Un petit front-end web vous est également fourni pour tester votre API. Vous n’avez qu’à ouvrir le fichier “index.html” dans votre navigateur web favori. Ce front-end fait un appel AJAX à votre API. Celle-ci doit donc évidemment être lancée.

Amusez-vous bien !

## 4 Exercices

### 4.1 Attributs (Fields)

Pour commencer, **importez le projet “introspect-api” dans votre IDE favori**. N’oubliez pas de mettre à jour le SDK si nécessaire. Maintenant, **observez ce qui vous est fourni** (*ignorez pour l’instant tout ce qui concerne les “instances”*) :

- La classe `Main` permet de démarrer le serveur web, et signale à Grizzly que l’API est dans le package `be.vinci.api`
- La classe `Classes` du package `be.vinci.api` répond à des requêtes GET sur l’URL `/classes`. Ceci est configuré grâce aux annotations au-dessus des méthodes et de la classe.
- Les routes de l’API font appel à la classe `services.ClassAnalyzer` qui semble faire le gros du “business” et qui renvoie directement du JSON.
- Pour envoyer du JSON, il faut passer par des `JsonObject`. Pour en créer, il faut passer par un `JsonObjectBuilder`. Au départ celui-ci est vide. On va ensuite le configurer en y ajoutant des choses (couples clef-valeur comme en JSON), et quand on a fini, on appelle la méthode `build()` sur le `JsonObjectBuilder`, qui va nous renvoyer un `JsonObject`.
- Les classes analysées doivent être placées dans le package `be.vinci.classes`.
- La classe `ClassAnalyzer` contient tout un tas de méthodes qui analysent la classe qui est stockée en attribut.

**Démarrez le serveur, et faites un appel pour observer le résultat.** Pour cela, copiez le lien suivant dans votre navigateur. Il fera alors un appel GET à l’URL que vous avez copiée.

<http://localhost:8080/reflect-api/classes>

**Démarrez le front-end** en ouvrant simplement le fichier `index.html` dans votre navigateur favori. Pour cela, double-cliquez dessus, ou glissez-déposez le dans votre navigateur. Une fois ouvert, il charge automatiquement la classe. Si ce n’est pas le cas, cliquez sur le bouton “Charger la classe” et observez le peu d’informations qu’il y a pour l’instant.

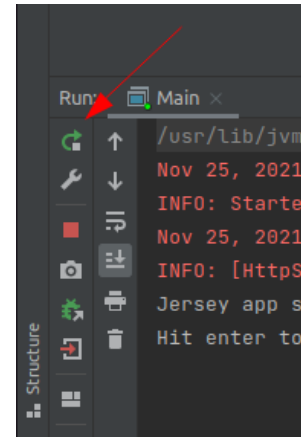
**Complétez les méthodes (les TODO) de la classe `ClassAnalyzer`** pour que les `Fields` soient correctement analysés et que le JSON voulu soit créé. **Lisez bien la JavaDoc des méthodes** qui vous explique le rôle de chaque méthode à compléter.

#### Quelques astuces :

- Seuls les attributs appartenant directement à la classe doivent être affichés. Ne tenez pas compte de l’héritage. Pour cela, **lisez bien la JavaDoc** des méthodes à votre disposition : `getFields()`, `getDeclaredFields()`...

- L'introspection fait usage des méthodes/classes... du **package java.lang.reflect**. Quand on vous propose un choix d'import, vérifiez bien que vous importez la bonne classe/méthode...
- N'hésitez pas à chercher sur le net lorsque vous rencontrez des difficultés à trouver votre bonheur dans la JavaDoc. Mais si vous avez un doute sur la solution, demandez aux professeurs.
- Si vous avez une erreur de type « Value in JsonObject's name/value pair cannot be null », c'est sans doute que vous n'avez pas encore implémenté la méthode `getFieldVisibility(Field f)`.
- Au fur et à mesure que vous complétez la classe, le JSON sera de plus en plus complet, et vous pourrez tester le résultat dans le front-end qui vous est fourni, et que vous avez ouvert précédemment, dans l'onglet "diagramme de classe".
- Vous pouvez relancer facilement votre serveur, pour prendre en compte vos changements, en cliquant sur le bouton "rerun" en bas à gauche.

Une fois terminé, **observez le JSON qui est renvoyé par le serveur à l'aide de l'application web**. Observez également le résultat graphique.



## 4.2 Méthodes (Methods)

**Complétez la classe ClassAnalyzer** pour qu'elle renvoie également les méthodes.

Pour cela, utilisez la même découpe en méthodes que pour les attributs. A savoir une méthode `getMethod()` et une autre `getMethods()` par exemple.

Voici le JSON qu'il faudrait renvoyer pour une méthode :

```
{
  name: "setFirstName",
  returnType: null,
  parameters: ["String"]
  visibility : "public"      // public, private, protected, package
  isStatic: false,
  isAbstract: false
}
```

Dans le JSON principal qui sera renvoyé au client, dans la méthode `getFullInfo()` de la classe `ClassAnalyzer`, il faudra ajouter ceci :

```
{
  methods: []                // Set methods here, same as Fields
}
```

Une fois terminé, **observez le JSON qui est renvoyé par le serveur à l'aide de l'application web**. Observez également le résultat graphique.

**!/ Attention** : ne complétez pas encore la classe `InstanceAnalyzer`, ce sera fait dans un exercice ultérieur.

**Astuce** : consultez la JavaDoc de la méthode `getParameters()` de la classe `Method`. Pour les paramètres, utilisez un `JSONArray`.

### 4.3 Choix de classe “dynamique”

Nous avons une API fonctionnelle pour analyser nos classes, générer une description en JSON de celles-ci, et les afficher ensuite sous forme graphique. Cependant, vous observerez que notre route d’API analyse systématiquement la même classe `User`. C’est un peu dommage, il faudrait pouvoir configurer cela “à la demande”.

Vous remarquerez que la méthode `getClassInfo` de la classe `Classes` prend un paramètre `classname`. Ce paramètre prend comme valeur le nom de la classe, qu’on peut passer en paramètre de la requête à l’API. Par exemple :

<http://localhost:8080/reflect-api/classes?classname=User>

Modifiez la méthode `getClassInfo`, sans toucher à sa signature, pour qu’elle utilise ce paramètre, et charge automatiquement la classe dont le nom est passé en paramètre, et se trouvant dans le package `be.vinci.classes`.

Si la classe n’existe pas, renvoyer une erreur 404 de cette manière :

```
throw new WebApplicationException(404);
```

### 4.4 Création du graphe d’instances

Un graphe d’instances est un ensemble d’instances qui ont des associations entre elles, et qui forment donc un graphe. Par exemple, un utilisateur qui référence sa liste de commandes, qui référence sa liste de lignes de commande. On a donc un graphe composé de tous ces objets qui ont des liens entre eux.

Pour nous simplifier la vie, nous allons travailler uniquement avec des graphes qui ne contiennent pas de cycles. Ils ressemblent donc plus à des arbres.

**Pour commencer, observez ce qui est fourni :**

- La création du graphe d’instances se fait à l’aide de “classes utilitaires” qui contiennent une unique méthode `initInstanceGraph()` qui renvoie un `Object`, racine de notre graphe d’instances. Un exemple est disponible dans le package `be.vinci.instances` : la classe `InstanceGraph1`
- La classe `Instances` dans le package `be.vinci.api` met à disposition une route “instances” qui va récupérer un graphe d’instances, l’analyser, et le renvoyer à l’appelant.

**Trouvez une manière de récupérer ce que renvoie actuellement la méthode `initInstanceGraph()`**, sans modifier les classes du package `be.vinci.instances`, et en utilisant le paramètre `builderclassname` qui correspond au nom de la classe du package `be.vinci.instances`, et sans appeler explicitement la méthode `initInstanceGraph()`.

Par exemple, si j’appelle l’URL suivante :

<http://localhost:8080/reflect-api/instances?builderclassname=InstanceGraph1>

La méthode `getInstanceGraphInfo()` doit récupérer le graphe d’instances généré par la méthode `initInstanceGraph()` de la classe `InstanceGraph1` se trouvant dans le package `be.vinci.instances`, et le stocker dans une variable locale de type `Object`.

Evidemment, ceci doit pouvoir fonctionner quel que soit la valeur du paramètre `builderclassname`.

Dans le cas où la classe n'existe pas, renvoyez également une erreur 404.

Dans le cas où la classe `InstanceGraph` est mal écrite (constructeur manquant, pas de méthode `init...`) renvoyez une `InternalError`.

**Astuces :** utilisez la méthode `"getMethod()"` pour récupérer la méthode et l'appeler. Certaines méthodes sont deprecated, ne les utilisez pas.

#### 4.5 Annotations

Actuellement, on se base sur le nom de la méthode `initInstanceGraph()` pour la récupérer et l'appeler.

Nous aimerions à présent nous détacher du nom de la méthode, pour que nous puissions avoir n'importe quel nom de méthode. Ceci va nous poser un souci, car la classe contient potentiellement plusieurs méthodes. Laquelle est celle qui crée le graphe d'instance complet ?

Pour l'identifier, nous allons la marquer à l'aide d'une annotation "personnalisée". Si ce n'est pas encore fait, veuillez prendre connaissance de la fiche de théorie sur les annotations.

**Créez une annotation `@InstanceGraphBuilder` dans un package `be.vinci.utils`, et utilisez-la** pour marquer la méthode `initInstanceGraph()` comme étant la méthode à appeler pour créer le graphe d'instances.

**Modifiez la méthode `getInstanceGraphInfo()` de la classe `Instances`** pour identifier la méthode à appeler à l'aide de l'annotation. Plus concrètement, il ne faut plus qu'on puisse trouver la chaîne de caractères `"initInstanceGraph"` dans le code de cette méthode.

#### 4.6 Analyse du graphe d'instances

**Complétez les méthodes de la classe `InstanceAnalyzer`** pour que les fields soient correctement analysés et que le JSON voulu soit créé. **Lisez bien la JavaDoc des méthodes** qui vous explique le rôle de chaque méthode à compléter.

**/!\ Attention**, dans un premier temps, ignorez les valeurs correspondant à des objets, et renvoyez `null` à la place dans le json. Dans ce cas, vous aurez un graphe incomplet, avec uniquement la racine. On le complètera dans un second temps.

Au fur et à mesure que vous complétez la classe, le JSON sera de plus en plus complet, et vous pourrez tester le résultat dans le front-end qui vous est fourni, et que vous avez ouvert précédemment, dans l'onglet "diagramme d'objets".

Une fois terminé, **observez le JSON qui est renvoyé par le serveur à l'aide de l'application web**. Observez également le résultat graphique.

#### 4.7 Bonus : Types génériques

Modifiez votre méthode `getField()` de `ClassAnalyzer` pour afficher les types génériques.

Voici ce que ça pourrait donner par exemple.

Ce n'est pas un souci si on voit le « fully qualified name » pour ces types là, mais il ne faut pas l'afficher pour les autres.

User
<u>+userIdSequence : int</u>
-id : int
-firstName : String
-lastName : String
-orders : java.util.List<be.vinci.classes.Order>

**Astuce** : renseignez vous sur : `ParameterizedType`.

#### 4.8 Bonus : graphe complet

**Modifiez la méthode `getField()` pour qu'elle gère également les attributs contenant des références.** Pour cela, instanciez un nouveau `InstanceAnalyzer` pour chaque instance, appelez-le, et récupérez le résultat que vous intégrez dans le JSON.

On a donc une forme de récursivité, car le `getField()` va créer un `InstanceAnalyzer` qui va appeler `getField()`...

**/!\ Attention**, lorsque le type est une liste, dans la clef value du JSON, il faut mettre comme valeur un tableau d'objets. L'application front-end n'affiche que la racine du graphe.