L API REFLECT

```
import java.lang.reflect.*;
```

- java.lang.reflect est une API qui permet principalement de connaître le contenu d'une classe de façon dynamique. L'introspection est un mécanisme très intéressant qui nous a permis lors de ce TP de factoriser du code.
- La classe Class ne possède pas de constructeur public mais il est possible de récupérer cette objet de plusieurs façons.
- Grâce à la méthode getClass(), nous pouvons récupérer la Class d'un objet. Nous pouvons par la suite récupérer toutes les informations que nous souhaiter sur cette classe :
  - getMethods() : renvoie toutes les méthodes publiques d'une classe.
  - getDeclaredMethods() : renvoie toutes les méthodes d'une classe.
  - Etc...

La classe Method

```
import java.lang.reflect.Method;
```

- L'API reflect nous offre une classe Method. Cette classe va nous permettre d'effectuer de nombreuses actions sur les méthodes d'une classe :
  - getAnnotation(Class annotationClass) : permet de récupérer une annotation.
  - getName() : retourne le nom de la méthode.
  - Invoke(Object obj, Object … args) : permet d'appeler la méthode.
  - Etc…

Les exceptions

```
private static String fieldAndFieldValue(Method e, Object obj){
        try {
            return getAnnotationNameOrMethodName(e) + ":" + e.invoke(obj);
        } catch (IllegalAccessException e1) {
            throw new IllegalStateException(e1);
        }
        catch (InvocationTargetException e2) {
            var cause = e2.getCause();
            if(cause instanceof RuntimeException)
                throw (RuntimeException)cause;
            if(cause instanceof Error)
                throw (Error)cause;
            throw new UndeclaredThrowableException(e2);
        }
}
```

- Lorsque nous devons gérer des exceptions, nous pouvons êtes confrontés à deux cas :
  - Les RuntimeException : nous ne devons pas les gérer
  - Et les autres exceptions : nous devons les gérer et l'exemple ci-dessus est un cas très fréquent que nous devons connaitre.
- IllegalAccesException fait référence à un objet privé qui n'est pas accessible.
- InvocationTargetException peut correspondre à trois type d'exceptions :
  - Une RuntimeException
  - Une Error
  - Et si ce n'est pas le cas, nous déclarons une UndeclaredRhrowableException()

Les annotations

```java
package fr.umlv.java.inside;
import java.lang.annotation.ElementType;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface JSONProperty {
    String value() default "";
}
```

- Nous avons vu qu'il était possible de créer nos propres annotations grâce à @interface.
- Quand nous créons une annotation, plusieurs choses sont à retenir :
  - Il est important de spécifier la Retention de l'annotation (@Retention); effectivement sans cela, notre annotation sera supprimer lors d'exécution du programme.
  - @Target nous permet de choisir où nous pouvons utiliser l'annotation, pour notre cas sur des méthodes.
  - Enfin, il est possible d'ajouter des valeurs (uniquement des primitives) aux annotations.

TravisCI : matrix

```yaml
language: java
jdk: openjdk13

# install:
#   - mettre une commmande si nécessaire

os :
  - linux
  - osx
env :
  matrix :
    - TARGET : "lab1"
    - TARGET : "lab2"
script :
  - cd $TARGET && mvn package
```

- Nous avons pu voir dans ce TP qu'il était possible de lancer plusieurs tests simultanément avec TravisCI. L'utilisation de matrix le permet.
- Nous créons dans matrix une variable TARGET qui aura deux valeurs différentes (une pour chaque projet) et dans le script principale du fichier yml nous appelons cette variable.
- Grâce à la fonctionnalité des matrix, nous allons avoir nos deux tests différents lancés parallèlement.

Le cache

```
private final static ClassValue<Function<Object, String>> cacheMethods = new ClassValue<Function<Object, String>>() {
    @Override
    protected Function<Object, String> computeValue(Class<?> type) {
        var methods = Arrays.stream(type.getMethods())
                            .filter(e -> e.getName().startsWith("get") && e.isAnnotationPresent(JSONProperty.class))
                            .sorted(Comparator.comparing(Method::getName))
                            .collect(Collectors.toList());

        return obj -> methods.stream()
                             .map(e -> fieldAndFieldValue(e, obj))
                             .collect(joining(",\n\t", "{\n\t", "\n}"));
    }
};
```

- En JAVA, il est possible de créer un cache sur certaines données. Ce cache nous permet de rendre l'exécution du code beaucoup plus rapide.
- Pour créer un cache, il faut utiliser la classe abstraite ClassValue<k>, k correspondant au retour de la méthode computeValue().
- cacheMethods.get() va nous permettre de faire appel à notre cache et deux cas se proposent à nous:
  - La valeur que nous cherchons est connue du cache, et aucun calcul n'est fait.
  - La valeur n'est pas connue et la méthode computeValue() est exécutée.
- Néanmoins, il faut savoir que certaines valeurs ne peuvent pas être mises en cache. Principalement celles qui ne sont pas final une fois le code lancé.