Java inside - lab 2 : **Reflection, Annotation, JSON**

SORAN Altan
Repository github: https://github.com/asoran/java-inside
Lab 2 : Reflection, Introspection, Annotation et ClassValue

Reflection

Le reflection c'est le fait de pouvoir examiner ou modifier le comportement de méthodes, classes et interface au runtime.

Le package propose une API import java.lang.reflect permettant de faire de la reflection en Java.
On peut notamment accéder aux méthodes et champs de la class, c'est ce qu'on a utiliser dans notre TP pour généraliser un comportement et éviter la répétition de code (et donc de bug).

Runtime vs Exception

Il faut bien faire la différence entre les RuntimeException et les Exception. De manière générale, la fonction qui génère l'exception ne sait pas gérer l'exception. C'est pour cela qu'il faut la propager.

Les RuntimeException, on ne les gère pas car elle se propage automatiquement, alors que le code ne compile pas si on ne gère pas les Exception "normales". Il faut essay RuntimeException pour gérer le cas.

Voici un gestion:

```
try {
    return m.invoke(that);

} catch (IllegalAccessException e) {
    throw new IllegalStateException(e);
} catch (InvocationTargetException e) {
    var cause = e.getCause();
    if(cause instanceof RuntimeException)
        throw (RuntimeException) cause;
    else if(cause instanceof Error)
        throw (Error) cause;
    throw new UndeclaredThrowableException(cause);
}
```

Annotations

Le annotations sont très utiles, et permettent aussi de généraliser des comportement mais aussi sert de documentation.

Il est possible de créer nos propres annotations grâce à @interface. On doit préciser des information sur l'annotation via des méta annotation:

@Target pour dire quelles types d'éléments on peut annoté et @Retention pour indiquer sa durée de vie. (On peut aussi mettre @Documented pour que l'annotation fait partie de la documentation de l'objet annoté)

Exemple:

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

}
```

Mettre en Cache

Certaines méthodes comme java.lang.Class.getMethods peuvent être très lente à cause par exemple de problème de mutabilité.
C'est pour cela qu'on peut utiliser la classe ClassValue pour sauvegarder des valeurs "dans une classe".
Pour cela, on crée un Object de type ClassValue<T> et on implémente la fonction 'computeValue' qui va renvoyer un Object de type T.

Exemple:

```
private final static ClassValue<Method[]> cachedMethods
    = new ClassValue<Method[]>() {
    @Override
    protected Method[] computeValue(Class<?> type) {
        // System.out.println("Test :)");
        return type.getMethods();
    }
};
```

Mettre en Cache

La fonction ‘computeValue’ prend en paramètre un objet Class, car en fait, à chaque apelle de la fonction avec une même classe, on aurait tout le temps la même chose.
Le contenue de la méthode est exécuté au premier appel, et est sauvegardé quelque part, les prochains appels ne vont pas exécuté la fonction et la fonction va directement renvoyé la valeur sauvegardé.

Il est important de noter que parfois, le code écrit dans ‘computeValue’ peut s’exécuter de nouveau. Par exemple, après 1 000 000 d’apelles à la fonction, j’observe que le cache s’est rafraîchit 3 fois.