Ingénierie du logiciel : cours 2

Frédéric Gava (d'après F. Martini)

Master ISIN, Université de Paris-Est Créteil

Cours Ingénierie du logiciel du M2 ISIN

Plan

Plan

Plan

Déroulement du cours

Introduction

Généralités

- Aussi appelé "MetaDonnées"
- Mieux documenter le code source ($\neq$ tag, p. ex @deprecated)
- Utilisation de ces informations pendant l'exécution
- Interagir avec le compileur APT : **A**nnotation **P**rocessing **T**ool
- Outil pour l'analyse formelle ($\Rightarrow$ sûreté/fiabilité)

Définition

Une annotation permet de "marquer" certains éléments du langage Java afin de leur ajouter une propriété particulière.

Une annotation se déclare comme une interface, mis à part le symbole @ devant le mot-clef **interface** qui permet de spécifier qu'il s'agit d'une interface :

public @interface MonAnnotation {...}

Placement

Utilisation

- Partout ⇒ package, class, interface, enum, constructeur, méthode, paramètre, champs d'une classe ou variable locale
- Autant qu'on en veut
- Mais pas 2 fois la même au même endroit !

```
@MonAnnotation1
@MonAnnotation2
public class MaClasse {...}
```

Des standards (1)

Heureusement, il existe des annotations standard. Par exemple @Deprecated qui signale à javac/javadoc que l'élément marqué est déprécié et ne devrait plus être utilisé. Javac affichera un warning si l'élément est utilisé dans du code non-déprécié mais pas la raison comme @deprecated.

Annotations standards (1)

Deprecated

```
public class Maclasse {
 /**
  * @return L'année en cours.
  * @deprecated Retourne en réalité le nombre d'années depuis 1900.
  * Remplacée par getFullYear(). */
  @Deprecated
  public int getYear() { return year;} ...}
```

Override

L'annotation ne peut être utilisée que sur une méthode afin de préciser au compilateur que cette méthode est redéfinie et doit donc "cacher" une méthode héritée d'une classe parent. Si ce n'est pas le cas, le compilateur échoue. Exemple :

```
@Override
public String toString() {return "Texte";}
```

Annotations standards (2)

SuppressWarnings

L'annotation permet de ne pas afficher certains warnings (parties crades/anciennes p. ex.). À utiliser avec parcimonie. Exemples :

```
@SuppressWarnings("deprecation") public class OldClass {...}
@SuppressWarnings({"deprecation","unckeked"}) public int m() {...}
```

Meta-annotations (annot d'annot de java.lang.annotation).

Documented

l'Annotation doit être présente dans la documentation générée (par javadoc) pour tous les éléments marqués.

```
public @interface SimpleAnnotation {}
import java.lang.annotation.Documented;
@Documented public @interface DocAnnotation {}
/** Commentaire de la méthode 1 */
@SimpleAnnotation public void method1 () {...}
/** Commentaire de la méthode 2 */
@DocAnnotation public void method2 () {...}
```

@SimpleAnnotation ne va pas apparaître. L'autre oui.

Annotations standards (3)

Inherit

L'annotation sera héritée par tous les descendants (que les classes) de l'élément sur lequel elle a été posée.

public @interface SimpleAnnotation {...}
import java.lang.annotation.Inherit;
@Inherit **public** @interface InheritAnnotation {...}
@SimpleAnnotation @InheritAnnotation **public class** ExempleInherited {...}

Retention (1)

Elle indique la "durée de vie" de l'annotation :

- RetentionPolicy.SOURCE ⇒ Les annotations ne sont pas enregistrées dans le fichier *.class. Elles ne sont donc accessibles que par des outils utilisant les fichiers sources
- RetentionPolicy.CLASS (**Par défaut**) ⇒ Les annotations sont enregistrées dans le fichier *.class maisne sont pas utilisées par la JVM (potentiellement par d'autres outils).
- RetentionPolicy.RUNTIME ⇒ Les annotations sont enregistrées dans le fichier *.class et utilisées par la JVM (réflection et introspection).

Annotations standards (4)

Retention (2)

```
@Retention(RetentionPolicy.SOURCE) @interface SourceAnnotation {}
@Retention(RetentionPolicy.RUNTIME) @interface RuntimeAnnotation {}
@SourceAnnotation @RuntimeAnnotation
public class Main {
  public static void main(String[] args) {
   for (Annotation a: Main.class.getAnnotations()) {
     System.out.println ("\t * Annotation : "
             +a.annotationType().getSimpleName()); } ...
```

Target

Elle permet de limiter le type d'éléments sur lesquels l'annotation peut être utilisée. Par défaut, partout. On trouve : ElementType.ANNOTATION_TYPE (peut être utilisée sur d'autres annotations), .CONSTRUCTOR, .FIELD, .LOCAL_VARIABLE, .METHOD, .PACKAGE, .PARAMETER, .TYPE (class, interface (annotation comprise) ou d'une énumération). Exemple :

```
@Target(ElementType.CONSTRUCTOR) public @interface ConstructorAnnotation {}
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD})
  public @interface ConstructorAnnotation {}
```

Déroulement du cours

Une simple annotation

Marqueur

```
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import static java.lang.annotation.RetentionPolicy.SOURCE;
@Documented @Retention(SOURCE) public @interface TODO {
    /** Message décrivant la tâche à effectuer. */
    String value(); <<=== toujours public }
@TODO(value="La gestion des exceptions est incorrecte...")
public void doSometing () {...}
```

Remarques :

- pour une annotation à membre unique, il dans le cas où il est possible de ne pas spécifier le nom de l'attribut lors de l'utilisation de l'annotation.
- Les annotations héritent implicitement de l'interface java.lang.annotation.Annotation
- Cas avec tableau :

```
@MonAnnotation (tab={1,2,3,4,5}) public class MaClasse {}
@MonAnnotation (tab=1) public class MaClasse {}
```

Marqueur (2)

Attention

Attributs possibles d'une annotation :

- type primitif (boolean, int, float, *etc.*).
- chaîne de caractères (java.lang.String).
- référence de classe (java.lang.Class).
- annotation (java.lang.annotation.Annotation).
- type énuméré (enum).
- tableau à une dimension d'un des types ci-dessus.

Membres et valeurs par défauts (1)

Niveau de priorité

Grâce à un type énuméré, nous allons enrichir notre annotation afin d'indiquer un niveau de priorité à la tâche @TODO :

```
@Retention(SOURCE) public @interface TODO {
 /** Message décrivant la tâche à effectuer. */ String value();
 /** Niveau de criticité de la tâche. */ Level level();
/** Enumération des différents niveaux de criticités. */
 public static enum Level { MINEUR, NORMAL, IMPORTANT }; }
```

Et son utilisation sera désormais (ordre indifférent) :

```
@TODO(value="Gestion exceptions incorrecte", level=TODO.Level.NORMAL)
/* ou */
import static com.developpez.adiguba.annotation.TODO.Level.*;
@TODO(value="Gestion exceptions incorrecte", level=NORMAL)
```

Valeur par défaut

... **public** @interface TODO { ... Level level() **default** Level.NORMAL; ... }

Et l'utilisation de level devient alors optionnel :

```
@TODO("Gestion exceptions incorrecte !")
@TODO(value="Gestion exceptions incorrecte !")
```

Membres et valeurs par défauts (2)

Plusieurs fois la même annotation

On ruse avec un tableau d'annotations :

```
@Documented @Retention(SOURCE) public @interface TODOs {TODO[] value();}
```

Qu'on utilise ainsi : :

```
@TODOs({
    @TODO("Gestion exceptions incorrecte !"),
    @TODO(value="NullPointerException possible.",level=IMPORTANT)})
```

Quelques trucs

Dans une annotation, on pourrait trouver ou faire :

- nom du développeur de la tâche ; date pour la réalisation
- **Eclipse** propose des TaskTag pour reporter certain mot-clefs dans les onglets
- **javadoc** pour traiter le tag @TODO dans les commentaires
- Utilisation de **apt** !

Déroulement du cours

Annotation Processing Tool

Qu'est-ce ?

Effectuer des traitement sur les annotations :

- Générer des messages : note, warning et error
- générer des fichiers :texte, binaire, sources/class Java
- API mirror de Sun (lib tools.jar). Doit être dans le classpath

Avant même de compiler le code Java ! On s'en sert comme **javac**.

Option en plus

- **-s dir** pour placer les fichiers générés
- **-nocompile**, **-print**
- **-A[key[=val]]**, paramètres pour les fabriques
- **-factorypath path**, indique un chemin autre que classpath
- **-factory classname**, **class** Java pour la "fabrique"

Utilisation (1)

Des fabriques

Elles permettent de créer les différents processus d'annotations.
Pour traîter des codes sources, 3 classes sont utilisées :

1. AnnotationProcessorFactory : la fabrique de bf apt
2. AnnotationProcessor : créee par la fabrique et utilisé par **apt** pour les fichiers sources
3. Visitors pour les différents déclarations d'un fichier source

Comment ça marche ?

apt utilisera des fabriques : fichiers nommés `com.sun.mirror.apt.AnnotationProcessorFactory` dans le répertoire META−INF/services des différents répertoire/archives jar : un simple fichier texte contenant le nom complet des différentes fabriques qui seront utilisées. Créons notre propre fabrique pour @TODO afin d'afficher des messages avec **apt**.

Une fabrique (1)

AnnotationProcessorFactory

Une interface pour créer une fabrique avec 3 méthodes :

1. supportedAnnotationTypes fournit la liste des annotations supportés.
2. supportedOptions fournit la liste des options acceptées (préfix "-A" doit être présent dans les valeurs retournées) de −Akey[=value]
3. getProcessorFor méthode appelée pour obtenir un objet AnnotationProcessor pour traîter les annotations. adts comporte la liste des annotations trouvées et env pour interagir avec **apt**

Exemple : SimpleAnnotationProcessor

Voir feuille. La classe SimpleAnnotationProcessor ⇒ notre processus.

Une fabrique (2)

AnnotationProcessor

Une interface avec une unique méthode : process qui le traitement des annotations. Pas de paramètre, don l'environnement **apt** doit lui être passé par la fabrique.

Exemple : SimpleAnnotationProcessor

Voir feuille. La méthode process() parcours la liste des déclarations de l'environnement d'**apt** et utilise la méthode accept() de ces déclarations pour les visiter. On utilise pour cela une méthode qui permet de scanner totalement une déclaration de type grâce au visitor.

Une fabrique (3)

DeclarationVisitor

Cette interface définit 16 (!) méthodes visit∗∗∗() permettant chacune de visiter une déclaration particulière. Lorsqu'on passe une instance de DeclarationVisitor à la méthode accept() d'une Declaration, la méthode visit∗∗∗() la plus appropriée est appelée. Pour nous simplifier, la classe SimpleDeclarationVisitor implémente toutes ces méthodes.

Exemple : TODOVisitor

Voir feuille. Notre instance de DeclarationVisitor utilisera seulement la méthode visitDeclaration() qui sera appelée pour n'importe quel type de déclaration. Notre visiteur affichera une note pour chaque annotation @TODO.

Compilation

```
import annotation.TODO;
import static annotation.TODO.Level.*;
public class Test {
  @TODO("Utiliser une annotation à la place d'une String")
  protected String day;
  @TODO(value="Ecrire le code de la methode", level=IMPORTANT)
  public void method() {} }
```

```
$apt -factory SimpleAnnotationProcessorFactory Test.java

 Note: day : Utiliser une annotation a la place d'une Strin
 Note: method : Ecrire le code de la methode
```

Autre exemple : un anti constructeurs vide

Dans de nombreux codes Java, des classes implémentent une interface particulière. La présence d'un constructeur vide est souvent requise et il était impossible jusqu'à présent de vérifier cela à la compilation. Corrigeont cela :

```
@Inherited
@Documented
@Target(ElementType.TYPE)
public @interface EmptyConstructor {}
```

Une simple annotation permet de signaler la présence d'un constructeur vide. **Voir feuille**. une fabrique pour traîter cela.

Restrictions

ANT

Vous pouvez utiliser APT malgré le fait qu'il ne propose pas de tâche <apt/>. Il suffit en effet d'utiliser la tâche <javac/> avec les attributs `fork=yes` et `executable=apt`. Par exemple :

```
<javac fork="yes" executable="apt" srcdir="${src}" destdir="${build}">
 <classpath>
  <pathelement path="tutoriel-annotations.jar"/>
 </classpath>
 <compilerarg value="-Arelease"/>
</javac>
```

Avec les EDI

Eclipse, NetBeans et JBuilder ne sont pas toujours bien compatible avec **atp** ... à tester.

A la semaine prochaine