C3I / NFP121

Programmation Avancée

Les Annotations en Java

Les Annotations

- Ce que sont les annotations
- Les annotations « livrées » avec le JDK et leur utilité
- Créer ses propres annotations
- Différents cas d'utilisation des annotations
- Exploiter les annotations

Qu'est ce qu'une @Annotation ? (1)

- C'est un élément textuel qui est ajouté dans le code source pour y ajouter des méta-informations
- Le nom de cet élément est préfixé par '**@**'
- Une annotation peut comporter des attributs valués

Exemples:

```
@Author(name = "AISL 2018")
public class MaClasse {
    //...
}
```

L'annotation `@Author` permet de préciser l'auteur de la classe.

Quelle différence avec le tag `@author` *de Javadoc ?*

Qu'est ce qu'une @Annotation ? (2)

Un autre exemple:

```
@Entity
@Table(name = "PointDeControle")
public class PointDeControle {
 @Id
 @GeneratedValue(strategy = GenerationType.SEQUENCE)
 public Long id;
 public String libelle;
 @Lob
 public String description;
 @OneToMany(targetEntity = QuestionImpl.class, mappedBy = "pointDeControle", fetch =
 FetchType.LAZY)
 public List<Question> questions = new ArrayList<Question>();
 // ...
```

Les annotations de javax.persistence *utilisées pour décrire la manière de réaliser le MOR (JPA)*

Les Annotations prédéfinies (fournies avec le JDK) (1)

Annotations applicables sur les éléments du code:

`@Deprecated:` L'élément annoté est obsolète et qu'il faudrait éviter de l'utiliser.

`@Override:` Pour indiquer qu'une méthode est la redéfinition d'une méthode d'une de ses super-classes. (Si faux une erreur de compilation se produit)

`@SuppressWarnings(« … »):` Pour demander au compilateur de ne pas produire certains avertissement pour la section de code sur laquelle est appliquée l 'annotation.

Depuis JSE 7:

`@SafeVarargs:` Pour indiquer qu'une méthode ne comporte aucune opération pouvant conduire à des erreurs de type.

Depuis JSE 8:

`@FunctionalInterface:` Pour indiquer qu'une interface est une interface fonctionnelle (pouvant être instanciée à la volée par une Lambda)

L’ Annotation prédéfinie @SafeVarargs (1)

```
@SafeVarargs

 // Méthode avec arguments variables d'un type connu
 public static void foo(String... args) {
  ◦ //...
 }

 // Pour appeler la fonction : void foo(String... args);
 foo("AA", "BB", "CC");

 // Java réécrit l'appel de la manière suivante:
 foo(new String[]{"AA", "BB", "CC"});
```

https://stackoverflow.com/questions/14231037/java-safevarargs-annotation-does-a-standard-or-best-practice-exist/14252221

L’ Annotation prédéfinie @SafeVarargs (2)

```
@SafeVarargs

 // Méthode avec arguments variables d'un type T inconnu
 public static <T> void foo2(T... args) {
  ◦ //...
 }

 // Si on utilise un paramètre générique de type T
 // Si le type de T est connu aucun problème (idem cas du String)
 // Mais si T n'est pas connu java devrait réécrire un appel de cette manière:
 // foo2(new T[]{...});
 // Mais new T[] n'est pas possible ni valide. donc java réécrit :
 // foo2(new Object[]{...});
 //
 // Si dans le code de Foo on suppose que le tableau constitué ne contient que des T : Pas de problème
 // Par contre si on considère que le tableau est un T[] alors que c'est un Object[] on aura un crash
 // avec une ClassCastException
```

L’ Annotation prédéfinie @SafeVarargs (3)

```
@SafeVarargs
 static <T> T[] asArray(T... args) {
     return args;
 }

 static <T> T[] arrayOfTwo(T a, T b) {
     return asArray(a, b);
 }
 // Si dans le code de Foo on suppose que le tableau constitué ne contient que des T : Pas de problème
 // Par contre si on considère que le tableau est un T[] alors que c'est un Object[] on aura un crash
 // avec une ClassCastException

 String[] arrString = arrayOfTwo("hi", "mom"); // ClassCastException !!!
```

Le compilateur n’est pas capable de déterminer si il y a un risque d’erreur ou non. Pour prévenir ce risque, le compilateur met systématiquement un avertissement (Warning). L’annotation `@SafeVarargs`
Permet d’annoncer au compilateur que nous savons que l’erreur ne se produira pas et qu’il ne doit pas créer de message d’avertissement.

Autres annotations prédéfinies (fournies avec le JDK) (2)

Une classe obsolète dans laquelle est redéfinie une méthode de sa superclasse :

```
@Deprecated
public class UneSimpleClasse extends SuperClasse {
	@Override
	public void annotatedMethod() {
	//...
	}
}
```

Une interface fonctionnelle :

```
@FunctionalInterface
public interface Lecteur {
	public void lire();
}
```

Autres annotations prédéfinies (fournies avec le JDK) (3)

Annotations applicables sur les Annotations:

`@Retention:` Pour définir jusqu'à quel niveau l'annotation doit être conservée

RetentionPolicy: SOURCE, CLASS, RUNTIME (CLASS par défaut)

`@Target:` Pour énumérer les types d'éléments sur lesquels l'annotation est applicables.

ElementType: ANNOTATION_TYPE, TYPE, LOCAL_VARIABLE, CONSTRUCTOR, FIELD etc.

`@Inherited:` Pour propager l'application d'une Annotation d'une super-classes vers ses sous-classes

`@Documented:` Indique que l'annotation doit apparaitre dans la Javadoc des éléments sur lesquels elle est appliquée.

Depuis JSE 8:

`@Repeatable:` Pour indiquer que l'annotation peut être appliquée plusieurs fois sur le même élément

Autres annotations prédéfinies (fournies avec le JDK) (4)

Annotations applicables sur les Annotations:

Depuis JSE 8:

`@Repeatable` : Pour indiquer que l'annotation peut être appliquée plusieurs fois sur le même élément.

```java
/**
 * Annotation pour Documentation
 * des Classes et interfaces
 */
@Documented
@Retention(RetentionPolicy.CLASS)
@Target({ TYPE})
@Repeatable(Infos.class)
public @interface Info {
    String auteur();
    String date();
    String version();
}
```

```java
/**
 * Container pour plusieurs
 * annoations Info
 */
@Documented
@Retention(CLASS)
@Target(TYPE)
public @interface Infos {
    Info[] value();
}
```

```java
// Répétition de l'annotation @Info
@Info(auteur = "PP", date = "12/10:2021", version = "0.1")
@Info(auteur = "PP", date = "05/10:2021", version = "1.0")
@Info(auteur = "PP", date = "11/10:2021", version = "1.01")
@Info(auteur = "PP", date = "13/10:2021", version = "1.02")
public class Personne {

    //...
}
```

Créer ses propres Annotations (1)

- Le mot clé **`@Interface`** permet de définir une annotation

```
public @interface MonAnnotation {
  // ...
}
```

- Les annotations **`@Retention`** et **`@Target`** sont à appliquer sur l'annotation pour définir: sa durée de vie et les éléments sur lesquels elle peut s'appliquer

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface MonAnnotation {
  // ...
}
```

Créer ses propres Annotations (2)

- Définition des **propriétés** pour une **annotation**
  - **Types autorisés: tous les types primitifs, String, Class, Enumération, Annotations, tableau de ces types**

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface MonAnnotation {
   String commentaire();
   double coeff();
   String[] directives();
}
```

```
// Application de MonAnnotation sur la classe UneClasse
@MonAnnotation(
    commentaire="à finaliser",
    coeff=0.5,
    directives={"TEST", "REVUE CODE"}
)
public class UneClasse {
   //...
}
```

  - **Nommage des propriétés:** le nom des propriétés respecte les mêmes règles que tous les éléments nommés du langage.

- Le nom des propriétés est suivi de **()**. Leur valeur sera **affectée en rappelant ce nom**.

- Si une annotation comporte **une seule propriété**, en la nommant **value** la valeur de cette propriété pourra être affecté sans rappeler son nom.

```
public @interface Remarque {
   String value();
}
```

```
@Remarque("à finaliser")
public class UneClasse {
   //...
}
```

Créer ses propres Annotations (3)

- **Valeur par défaut d'une propriété:** définition d'une valeur par défaut avec le mot clé **`default`**

```
public @interface MonAnnotation {
   String commentaire();
   double coeff() default 0.25; // Valeur par défaut pour coeff
   String[] directives();
   Auteur auteur(); // propriété de type Annotation
}
```

```
public @interface Auteur {
   String name();
}
```

```
// Application de MonAnnotation sur la classe UneClasse
// (coeff a une valeur par défaut et devient optionelle)
@MonAnnotation(
    commentaire="à finaliser",
    directives={"TEST", "REVUE CODE"},
    auteur= @Auteur(name="AISL") // la valeur est une annotation
)
public class UneClasse {
   //...
}
```

A quoi servent les annotations ? (1)

- Elles sont des métadonnées appliquées sur les éléments du code :

Classe, Interface, Champ, Méthode, Paramètre, Constructeur, Variable, Annotation, Package

- Elles permettent d'ajouter des informations complémentaires pour caractériser les éléments d'un code Java.

Simples informations, Marquage, Expressions de contraintes, Configuration pour fmk etc.

- Elles sont exploitées à différentes phases du cycle de vie du code par :

Les humains lecteurs du code, Les processeurs d'annotations (phase de pré-compilation), Le compilateur, Le chargeur de classe de la JVM, Le code exécuté.

A quoi servent les annotations ? (2)

- **L'exploitation des métadonnées fournies par les annotations permet de :**

1) Fournir des informations au lecteur d'un code
2) Générer des artefacts ou du code (durant la phase de précompilation)
3) Effectuer certaines validations à la compilation
4) Modifier du code à la volée pendant l'exécution (lecture de classe)
5) Agir pendant l'exécution (analyse des annotations par introspection)

1) Annotations pour fournir des informations

- Informations destinée au lecteur d'un code :

```
@Author(name = "AISL 2018")
public class UneSimpleClasse {
      @Deprecated
      public void annotatedMethod() {
      //...
      }
}
```

`@Author` permet de spécifier l'auteur de la classe (Annotation personnalisée)

`@Deprecated` permet d'indiquer que la méthode est dépréciée et ne devrait pas être utilisée. L'IDE peut l'utiliser pour mettre en évidence cette information. C'est une annotation livrée avec le JDK

2) Annotations pour générer des artefacts (1)

- Génération d'artefacts ou de code source durant la phase de précompilation :

Exemple : Génération d'un service locator à partir des annotations @AISLService

```
@AISLService
public class ServiceTutu implements ServiceAvecTest{
 public void test(){
        System.out.println("Je suis un service tutu");
 }
}

@AISLService
public class UnAutreService implements ServiceAvecTest{
 //...
}
```

`@AISLService` *permet d'indiquer qu'il s'agit d'un service devant être référencé par le* ***ServiceLocator****. Le processeur d'annotation traite les annotations @AISLService et les utilise pour générer le code Java de la classe ServiceLocator.*

2) Annotations pour générer des artefacts (2)

```
//Service locator Generated by AISL Annotation processor on Mon Sep 03 10:56:04 CEST 2018
public class ServiceLocator{
 //Singleton
 protected static ServiceLocator instance = new ServiceLocator();
 public static ServiceLocator getInstance(){return instance;}
 private ServiceLocator(){}

 protected aisl.fmk.demo.ServiceTutu serviceTutu;
 public void setServiceTutu (aisl.fmk.demo.ServiceTutu serviceTutu){this.serviceTutu = serviceTutu;}
 public aisl.fmk.demo.ServiceTutu getServiceTutu(){return this.serviceTutu;}
 //...
```

*Un **processeur d'annotation enregistré auprès du compilateur** est invoqué chaque fois qu'une classe annotée `@AISLService` est rencontrée.*

2) Annotations pour générer des artefacts (3)

- Génération d'artefacts ou de code source durant la phase de précompilation ou à l'exécution :

Un autre exemple : les annotations JPA pour décrire le mapping et créer les tables de la Bdd

```
@Entity
@Table(name = "PointDeControle")
public class PointDeControleImpl {
 @Id
 @GeneratedValue(strategy = GenerationType.SEQUENCE)
 public Long id;
//…
```

Annotations de JPA (javax.persistence) destinées à une implémentation d' ORM (EclipseLink, Hibernate) permettant de générer le script SQL pour créer les tables dans une Bdd relationnelle.

3) Validations / Contrôles à la compilation (1)

- Le compilateur vérifie que les « règles » ou « assertions » exprimées par des annotations sont respectées :

```
public class SuperClasse {
  public void controler() {
  //...
  }
  public void traiter() {
  //...
  }
}
```

```
public class SousClasse extends SuperClasse
{
  @Override
  public void traiter() {
  //...
  }
}
```

L'annotations **`@Override`** appliquée sur la méthode **`traiter`** de la classe **`SousClasse`** impose que cette méthode redéfinisse la méthode **`traiter`** de **`SuperClasse.`**

3) Validations / Contrôles à la compilation (2)

```
public class SuperClasse {
  public void controler() {
  //...
  }
  public void traiter(String str)
  {
  //...
  }
}
```

```
public class SousClasse extends SuperClasse
{
  @Override
  public void traiter() {
  //...
  }
}
```

La méthode **`traiter`** de la classe **`SousClasse`** ne redéfinit pas la méthode **`traiter`** de **`SuperClasse.`** La présence de l'annotation **`@Override`** sur la méthode provoque une erreur à la compilation :

"The method traiter() of type SousClasse must override or implement a supertype method"

4) Modification / Injection de code au chargement de la classe

- Des classes annotées peuvent être « Instrumentées » pour leur ajouter un nouveau comportement ou agir sur leur comportement :

Exemples d'une méthode transactionnelle:

```
public class ServiceTransactionnel {
  @TransactionAttribute(REQUIRED)
  public void faireLeTruc() {
  //...
  }
}
```

L'annotations **`@TransactionAttribute`** sert à marquer la méthode comme transactionnelle. Au chargement de la classe du code est injecté pour encadrer le code de la méthode avec des instructions permettant d'obtenir le comportement transactionnel attendu .

5) Annotations pour être utilisées pendant l'exécution (1)

- Des éléments annotées sont porteurs d'informations qu'exploitent le code en cours d'exécution pour agir sur son déroulement.

- La recherche des annotations se fait en utilisant la reflection (`java.lang.reflect`)
  - Seules les Annotation de « Rétention » Runtime peuvent être retrouvés dans une Classe
  - Les annotations sont des types (presque) comme les autres il existe un .class pour chacune
  - Les annotations peuvent être retrouvées sur tous les types d'éléments :

```
Les Classes: java.lang.Class
Les Méthodes: java.lang.reflect.Method
Les Champs: java.lang.reflect.Field
...
```

5) Annotations pour être utilisées pendant l'exécution (2)

- A l'exécution du code fouille les éléments et recherche les annotations et effectue leur traitement

```
// Fouille d'une classes et de ses éléments
Class<?> clazz = ... ;
clazz.getInterfaces() // liste les interfaces implémentées par la classe
clazz.getDeclaredFields()   // liste des champs membre de la classes
clazz.getDeclaredMethods() // liste des méthodes membre de la classes
method.getParameters()      // liste des paramètres d'une méthode

// Les éléments annotables implémentent l'interface AnnotatedElement
java.lang.reflect.AnnotatedElement:
Annotation[] getDeclaredAnnotations(); // Annotations sur l'élément
boolean isAnnotationPresent(Class<? extends Annotation> annotationClass) // test présence annotation
```

https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/AnnotatedElement.html

Cas d'utilisation: Test (Junit), Injection de dépendance (Guice), ORM (JPA)

5) Annotations utilisées pendant l'exécution : Démo (1)

- Démo « Le Vérificateur d'obsolescence »
  - Fouille d'une classe et de tous ses éléments pour découvrir ses annotations:

Classe, Interfaces implémentées, Propriétés, Méthodes, Types internes

```
public void verifierClasse(Class<?> clazz) {
    // Contrôle de la classe
    controlerObsolescence(clazz, clazz);
    // Contrôle des interfaces implémentées
    for(Class<?> implementedInterface : clazz.getInterfaces()) {
        controlerObsolescence(clazz, implementedInterface);
    }
    // Contrôle des champs
    for(Field field : clazz.getDeclaredFields()) {
        controlerObsolescence(clazz, field);
        controlerObsolescence(clazz, field.getType());
    }

    // ...
}
```

5) Annotations utilisées pendant l'exécution : Démo (2)

- Démo « Le Vérificateur d'obsolescence »
  - Recherche de la présence d'annotations `@Deprecated` :

```
/**
 * Vérifie si l'élément est annoté @Deprecated
 * Si il l'est un message d'avertissement est affiché
 * @param annotated
 */
protected void controlerObsolescence(Class<?> clazz, AnnotatedElement annotated) {
    if(annotated.isAnnotationPresent(Deprecated.class)) {
        System.out.println("ATTENTION: la classe " + clazz.getName()
        + " utilise l'élément deprecated " + annotated );
    }
}
```

Conclusion

- Applications normées d'informations (à la place de commentaires)
- Un langage au-dessus de Java (Peut aussi nuire à la lisibilité)
- Améliore la qualité du code: Définition de contraintes pouvant être contrôlées à la compilation
- Couplées à des Framework ou outils elles ajoutent des fonctionnalités :
  - Elles induisent des comportements sans intervention du programmeur (IOC)
  - Elles permettent de configurer et d'utiliser des fonctions évoluées de frameworks