Les annotations

Les annotations en Java sont des marqueurs qui permettent d'ajouter des méta-données aux classes, aux méthodes, aux attributs, aux paramètres, aux variables, aux paquets ou aux annotations elles-mêmes.

Les annotations sont utilisées dans des domaines divers. Leur intérêt principal est de fournir une méta-information qui pourra être exploitée par un programme.

Utilisation des annotations

Une annotation est un type (comme une classe ou une interface) du langage Java : elle peut être référencée par son nom complet ou importée depuis un autre paquet grâce au mot-clé **import**.

Une annotation n'est pas instanciée, elle est simplement accolée à l'élément qu'elle vient enrichir :

```
package com.cgi.udev;


public class Voiture {


  @Override
  public String toString() {
    return "une voiture";
  }


}
```

L'annotation Override est définie dans le package java.lang (c'est pour cela qu'il n'est pas nécessaire de l'importer explicitement). Cette annotation est utilisable uniquement sur les méthodes pour indiquer que la méthode est une redéfinition d'une méthode d'une classe parente (dans l'exemple précédent, la méthode redéfinit Object.toString). Cette annotation est exploitée par le compilateur pour réaliser des vérifications supplémentaires. C'est également le cas pour les autres annotations déclarées dans le même package :

Deprecated
: Permet de générer des warnings afin d'informer les autres développeurs que quelque chose (une classe, une méthode…) a été dépréciée et ne devrait plus être utilisée.

FunctionalInterface

Permet au compilateur de s'assurer que l'interface qui porte cette annotation peut être implémentée par une lambda (Cf. le chapitre sur les lambdas).

Override
: Signale qu'une méthode est une redéfinition d'une méthode déclarée dans une classe parente. Cela permet au compilateur de signaler une erreur si ce n'est pas le cas.

SuppressWarnings
: Permet de forcer le compilateur à ne plus émettre d'avertissement à la compilation dans certains cas.

SafeVarargs
: Cette annotation s'ajoute à une méthode acceptant un paramètre variable (*varargs*) dont le type est un générique. En effet, le principe de l'effacement de type (*type erasure*) dans la gestion des classes génériques fait qu'il est possible de corrompre un type paramétré utilisé comme paramètre variable sans que le compilateur et la JVM ne puissent le détecter. Pour pallier à ce problème, le compilateur produit systématiquement un avertissement lorsqu'on utilise un type générique comme paramètre variable. Cette annotation permet de supprimer l'avertissement à la compilation et implique que le développeur s'est assuré que son implémentation est sûre.

L'API standard de Java (mais également des bibliothèques tierces) fournissent beaucoup d'autres annotations qui ne sont pas interprétées par le compilateur mais par le programme lui-même à l'exécution.

Certaines annotations déclarent des attributs. Il est possible de spécifier entre parenthèses la valeur de chaque attribut d'une annotation. Par exemple, l'annotation XmlRootElement permet d'indiquer qu'une classe peut être instanciée à partir d'un document XML et/ou qu'une de ses instances peut servir à générer un document XML. Cette annotation accepte deux attributs optionnels : *name* pour donner le nom de l'élément XML correspondant et *namespace* pour donner l'espace de nom XML auquel l'élément appartient.

```
package com.cgi.udev;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "personne", namespace="http://xml.personne.com/ns")
public class Personne {

  private String prenom;
  private String nom;

  // ...

}
```

Si un attribut est de type tableau alors, il est possible de passer plusieurs valeurs entre accolades :

```
@SuppressWarnings(value = { "deprecation", "unused" })
public void doSomething() {
  // ...
}
```

Mais si un attribut est de type tableau et que l'on veut fournir une seule valeur alors, les accolades peuvent être omises :

```
@SuppressWarnings(value = "unused")
public void doSomething() {
  // ...
}
```

Enfin, si l'attribut porte le nom spécial **value** et qu'il est le seul dont la valeur est donnée alors, il est possible d'omettre le nom :

```
@SuppressWarnings("unused")
public void doSomething() {
  // ...
}
```

Déclaration d'une annotation

Comme pour les classes, les interfaces et les énumérations, on crée une annotation dans un fichier portant le même nom que l'annotation avec l'extension *.java*. On déclare une annotation avec le mot-clé **@interface**.

```
package com.cgi.udev;


public @interface MyAnnotation {


}
```

> Une annotation implémente implicitement l'interface Annotation et rien d'autre !

La déclaration des attributs d'une annotation a une syntaxe très particulière :

```
package com.cgi.udev;


public @interface MyAnnotation {
  String  name();
  boolean isOk();
  int[]   range() default {1, 2, 3};
}
```

Les attributs d'une annotation peuvent être uniquement :

- un type primitif,
- une chaîne de caractères (java.lang.String),
- une référence de classe (java.lang.Class),
- une Annotation (java.lang.annotation.Annotation),
- une énumération,
- un tableau à une dimension d'un de ces types.

Le mot-clé **default** permet de spécifier une valeur d'attribut par défaut si aucune valeur n'est donnée pour cet attribut lors de l'utilisation de cette annotation.

La déclaration d'une annotation peut elle-même être annotée par :

Documented
: Pour indiquer si l'annotation doit apparaître dans la documentation générée par un outil comme *javadoc*.

Inherited
: Pour indiquer que l'annotation doit être héritée par la classe fille.

Retention
: Pour préciser le niveau de rétention de l'annotation (Cf. ci-dessous).

Target
: Pour indiquer quels types d'éléments peuvent utiliser l'annotation : classe, méthode, attribut...

Repeatable
: Pour indiquer qu'une annotation peut être déclarée plusieurs fois sur un même élément.

```
package com.cgi.udev;


import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
```

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Documented
@Target(ElementType.TYPE)
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
  String  name();
  boolean isOk();
  int[]   range() default {1, 2, 3};
}
```

L'annotation ci-dessus porte des méta-annotations qui indiquent que l'utilisation de cette annotation doit apparaître dans la documentation générée, qu'elle est utilisable sur les types Java (c'est-à-dire les classes, les interfaces) et que sa rétention est de type *RUNTIME*.

Rétention d'une annotation

Une annotation est définie par sa rétention, c'est-à-dire la façon dont une annotation sera conservée. La rétention est définie grâce à la méta-annotation Retention. Les différentes rétentions d'annotation sont :

SOURCE
: L'annotation est accessible durant la compilation mais n'est pas intégrée dans le fichier class généré.

CLASS
: L'annotation est accessible durant la compilation, elle est intégrée dans le fichier class généré mais elle n'est pas chargée dans la JVM à l'exécution.

RUNTIME
: L'annotation est accessible durant la compilation, elle est intégrée dans le fichier class généré et elle est chargée dans la JVM à l'exécution. Elle est accessible par introspection.

Utilisation des annotations par introspection

Une annotation ne produit aucun traitement. Cela signifie que si on utilise des annotations dans son code, encore faut-il qu'un processus les interprète pour produire le comportement attendu. Hormis les quelques annotations interprétées par le compilateur, il faut donc s'assurer que les annotations seront traitées correctement.

Pour des annotations de rétentions **SOURCE** et **CLASS**, leur interprétation dépend de processeurs d'annotations qui sont des bibliothèques Java déclarées en paramètre du compilateur ou de la JVM. Il s'agit d'une utilisation assez avancée et relativement peu utilisée (en dehors des annotations directement prises en charge par le compilateur lui-même).

> Lombok est un exemple de projet open-source fournissant des annotations permettant de générer du code au moment de la compilation grâce à un processeur d'annotations.

L'utilisation la plus courante (notamment avec Java EE) est l'utilisation d'annotation de rétention **RUNTIME** car elles sont accessibles par introspection.

Java fournit une API standard appelée l'API de réflexion qui permet de réaliser à l'exécution une introspection des objets et des classes. Cela signifie qu'il est possible de connaître par programmation tout un ensemble de méta-informations. Par exemple, on peut connaître la liste des méthodes d'une classe et pour chacune le nombre et le type de ses paramètres. Mais surtout, on peut connaître les annotations utilisées et la valeur de leurs attributs.

Imaginons que nous souhaitions créer une framework de tests automatisés. Nous pouvons créer l'annotation *@Test* qui servira à indiquer quelles méthodes publiques d'une classe correspondent à des tests à exécuter par notre framework.

```
package com.cgi.udev.framework.test;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Documented
@Inherited
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
}
```

Comme la rétention de cette annotation est **RUNTIME**, il est possible d'accéder à cette annotation par introspection. Le framework de test peut contenir une classe *TestFramework* qui accepte une instance de n'importe quel type d'objet et qui va exécuter une à une les méthodes publiques ayant l'annotation *@Test*.

```
package com.cgi.udev.framework.test;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class TestFramework {

  public static void run(Object o) {
    Method[] methods = o.getClass().getMethods();
    for (Method method : methods) {
      if (method.isAnnotationPresent(Test.class)) {
        runTest(o, method);
      }
    }
  }

  private static void runTest(Object o, Method method) {
    try {
      method.invoke(o);
      System.out.println("Test " + method.getName() + " ok");
    } catch (InvocationTargetException e) {
      System.err.println("Test " + method.getName() + " ko");
      e.getTargetException().printStackTrace();
    } catch (Exception e) {
      System.err.println("Test " + method.getName() + " ko");
      e.printStackTrace();
    }
  }
}
```

Grâce à l'API de réflexion, il est possible d'accéder à la représentation objet d'une classe avec la méthode getClass.

Finalement, nous pouvons écrire une pseudo-classe de tests :

```
package com.cgi.udev;
```

```
import com.cgi.udev.framework.test.Test;
import com.cgi.udev.framework.test.TestFramework;

public class MesTests {

  @Test
  public void doRight() {
    // ...
  }

  @Test
  public void doWrong() throws Exception {
    // ...
    throw new Exception("simule un test en échec");
  }

  public static void main(String[] args) {
    TestFramework.run(new MesTests());
  }
}
```

Revision #1
Created 2025-02-10 23:29:36 UTC by Nicolas
Updated 2025-02-10 23:30:48 UTC by Nicolas