Les annotations

Contenu

Introduction

Une annotation Java est une façon d'ajouter des méta-données à un code source Java. Elles peuvent être ajoutées aux classes, méthodes, attributs, paramètres, variables locales et packages. Elles permettent non seulement de mieux documenter le code source, mais également d'utiliser ces informations pendant l'exécution et même d'interagir avec le compilateur. Les annotations sont utilisées par des composants extérieurs qui manipulent ce code.

Une annotation ne change pas directement la sémantique d'exécution du programme. Elle sert d'**instructions complémentaires** à des outils et librairies afin de modifier le comportement, ce qui au final peut changer la sémantique d'exécution du programme.

Concrètement, une annotation commence par `@` et se place **avant la signature** de méthode, la déclaration de la classe, la déclaration d'un champ, d'une variable locale ou d'un paramètre ... Plusieurs annotations différentes peuvent être utilisées sur un même élément mais on ne peut pas utiliser deux fois la même annotation. En général l'annotation est placée devant la déclaration de l'élément qu'elle marque. On ne met pas de ; à la fin d'une annotation.

Les annotations standards sont `@Override`, `@Deprecated` et `@SuppressWarnings`.

Exemples:

```
@Override
public String toString() {…}
```

```
@Deprecated
public Date(int a, int m, int j){ … }

@SuppressWarnings("unchecked")
public void LegacyCode(ArrayList a){ … }
```

Les types d'annotation

Il existe différents types d'annotations :

- Marker Annotation : elles ne possèdent pas de membres.

  ```
  @Override @Deprecated
  ```

- Single-value Annotation : elles possèdent un seule membre qui est `value`.

  ```
  @SuppressWarnings("unchecked")
  ```

  `value` dans l'exemple prend la valeur `unchecked`.

- Full Annotation : elles possèdent plusieurs membres nommés.

  ```
  @Correction(resultat = Correction.BIEN, correcteur =
  "Steph", commentaire = "2h")
  ```

@Override

`@Override` permet d'éviter des erreurs typographiques dans le nom des méthodes redéfinies. L'annotation `@Override` précise au compilateur que la méthode annotée est redéfinie et doit donc 'cacher' une méthode héritée d'une classe parent. Si ce n'est pas le cas (par exemple si une faute de frappe s'est glissée dans le nom de la méthode), le compilateur doit lancer une erreur car il s'agit alors d'une simple définition.

Par exemple, lorsqu'on redéfinit la méthode `toString()` de `Object` :

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

@SuppressWarnings

`@SuppressWarnings` indique au compilateur de ne pas afficher certains warnings. Elle indique qu'il ne faudra pas générer d'avertissement du type "`rawtypes`", c.-à-d. qu'on n'utilise pas les generics avec le type `Class`.

`@SuppressWarnings` a un membre nommé value et de type `String[]`.

Quelques exemples :

```
@SuppressWarnings({ "unused", "rawtypes" })
private List uneListe;
```

L'annotation m'informe qu'`uneListe` n'est pas utilisé et qu'elle utilise un type `List` qui devrait être générique `List<T>`.

@Deprecated

`@Deprecated` permet de signaler au compilateur que l'élément marqué ne devrait plus être utilisé (est déprécié). On obtient une erreur de compilation (si on a configuré Eclipse dans ce sens : **Windows – Preferences – Java – Compiler – Error/Warnings – J2SE 5.0 Options – Missing @Deprecated annotation**). Attention, il existe également un tag Javadoc `@deprecated` qui, lui, n'est pas interprété par le compilateur.

Par exemple, la méthode de classe `parse` de la classe `Date` est dépréciée mais on peut encore l'utiliser.

```
@Deprecated
public static long parse(String s)
```

Si on décide de l'utiliser quand même,

```
@SuppressWarnings("deprecation")
Long uneDate = Date.parse("Sat, 12 Aug 1995 13:30:00 GMT");
```

Usage des annotations

On peut annoter une classe, une méthode, un attribut, un paramètre ou une variable. L'annotation se fait souvent sur la ou les lignes précédant l'élément à annoter, ou sur la même ligne avant l'élément. Une annotation commence toujours par @ suivi d'un nom d'annotation suivi éventuellement de paramètres entre parenthèses. Les annotations peuvent être de 3 types et c'est l'annotation elle-même qui choisit son type :

- Disponible uniquement à la compilation : comme dans les exemples `@Override` et `@SuppressWarnings` ci-dessus, ces annotations sont destinées à changer le comportement du compilateur. A l'exécution, on n'aura aucune trace de ces annotations. C'est donc uniquement les concepteurs du compilateur qui sont concernés par ces annotations.

- Disponible à la compilation et au déploiement : en plus d'être disponibles pour le compilateur, ces annotations sont disponibles aux outils qui analysent le code (par exemple générer des fichiers XML qui configurent un web service). Elles sont donc présentes dans les fichiers .class, mais absentes à l'exécution. C'est donc uniquement les concepteurs d'un outil d'analyse de bytecode qui sont concernés par ces annotations.
- Disponible à l'exécution : par introspection on peut obtenir l'information d'annotation directement à l'exécution. Les classes `Class`, `Method` et `Field` possèdent toutes une méthode `getDeclaredAnnotations()` qui retourne un tableau d'`Annotation`, confer la javadoc. En général on crée uniquement des annotations de ce type, voir ci-dessous.

Création d'annotation

Java permet de créer ses propres annotations et l'introspection permet de récupérer les informations d'annotations. Voyons cela plus en détails.

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


@Retention(RetentionPolicy.RUNTIME)
public@interfaceMonAnnotation {

}
```

Créer une annotation est similaire à créer une interface, mais notez la présence du @ avant le mot `interface`. Le `@Retention(RetentionPolicy.RUNTIME)` permet de signaler que le type de cette annotation est d'être disponible à l'exécution. On pourra donc dorénavant annoter un élément par `@MonAnnotation`, l'introspecter et utiliser la méthode `getDeclaredAnnotations()` pour récupérer l'annotation. Ou pourra aussi introspecter un élément et appeler `isAnnotatedPresent(MonAnnotation.class)` pour déterminer si cet élément est annoté par `MonAnnotation` ou pas.

On peut aussi créer des annotations avec état :

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

@Retention(RetentionPolicy.RUNTIME)
Public @interface MonAnnotationAvecEtatSimple {
      int value();
}
```

Dans cette version simplifiée, l'annotation reçoit un seul et unique paramètre. Le type de ce paramètre est déterminé par le type de retour de la fonction `value()`. **Les types autorisés pour les annotations**

sont restreints aux types primitifs, aux chaînes de caractères, au type Class, aux énumérés, aux annotations et aux tableaux des types précédents.

```
@MonAnnotationAvecEtatSimple(10)
Public class TestAnnotation {

	public static void main(String[] args) {
		MonAnnotationAvecEtatSimple m = TestAnnotation.class
				.getAnnotation(MonAnnotationAvecEtatSimple.class);
		System.out.println(m.value());
	}
}
```

Si on exécute cette classe, elle affichera 10 à la console. L'annotation est récupérée à l'exécution par la méthode `getAnnotation`. Notez aussi comment l'annotation est utilisée comme une interface, c'est d'ailleurs de cette manière qu'on récupère la valeur qui y est stockée.

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


@Retention(RetentionPolicy.RUNTIME)
public @interface MonAnnotationAvecEtat {
	String uneChaine();
	int compteur() default 1;
	String date();
}
```

Une telle annotation s'utilise comme ceci :

```
import java.lang.annotation.Annotation;

@MonAnnotationAvecEtat(
		date="hier",
		uneChaine="salut monde"
)
public class TestAnnotation {
	public static void main(String[] args) {
		MonAnnotationAvecEtat aa=TestAnnotation.class
				.getAnnotation(MonAnnotationAvecEtat.class);
		System.out.println(aa.compteur());
		System.out.println(aa.date());
		System.out.println(aa.uneChaine());
	}
}
```

Et ce programme affichera:

```
1
hier
salut monde
```

Notez que l'on n'a pas spécifié la valeur de compteur et que son `default` a donc été utilisé.