Annotations en Java

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-11-01

Table des matières

1 Annotations en Java : Des métadonnées pour votre code 1
2 Annotations standards 2
3 Définition d'une annotation 2
4 Paramètres d'une annotation 3
5 Value 3
6 Portée d'une annotation 4
7 Annotations répétables 1/2 5
8 Annotations répétables 2/2 5
9 Annotations de package 6
10 Durée de vie et héritage des annotations 6
11 Utilisation par instrospection (1/2) 7
12 Utilisation par instrospection (2/2) 8

1 Annotations en Java : Des métadonnées pour votre code

Les annotations en Java sont des **métadonnées** que l'on associe à différentes entités du code (classes, méthodes, attributs, etc.) afin de fournir des informations supplémentaires à propos de celui-ci.

- À quoi servent les annotations ?
  - **Documentation:** Améliorez la lisibilité de votre code en fournissant des explications contextuelles.
  - **Configuration:** Paramétrez le comportement de vos applications sans modifier le code source.
  - **Traitement du code source:** Automatisez des tâches comme la génération de code, la création de tests ou la validation de contraintes.
  - **Intégration avec des outils:** Facilitez l'utilisation de frameworks et d'outils externes.
- Syntaxe de base

- Une annotation se définit avec le caractère `@` suivi de son nom. Par exemple :

```
@SuppressWarnings("unused")
private int maVariable;
```

2 Annotations standards

Il existe plusieurs annotations standards couramment utilisées en Java :

- `@Deprecated` (Lien vers la documentation d'Oracle pour @Deprecated) : Signale qu'une classe, méthode, etc. est déconseillée et qu'il est préférable d'utiliser une alternative.
- `@FunctionalInterface` (Lien vers la documentation d'Oracle pour @FunctionalInterface) : Indique qu'une interface peut être utilisée pour la programmation fonctionnelle, ne pouvant avoir qu'une seule méthode abstraite.
- `@Override` (Lien vers la documentation d'Oracle pour @Override) : Spécifie qu'une méthode redéfinit une méthode héritée d'une classe parente.
- `@SuppressWarnings` (Lien vers la documentation d'Oracle pour @SuppressWarnings) : Demande au compilateur d'ignorer certains types d'avertissements pour un élément de code spécifique.
- `@SafeVarargs` (Lien vers la documentation d'Oracle pour @SafeVarargs) : Indique que la méthode utilisant des arguments de type générique ne provoque pas de problèmes de sécurité de type au moment de l'exécution.

3 Définition d'une annotation

Une annotation se déclare dans un fichier qui a le même nom comme une classe ou une interface.

```
public @interface MyAnnotation { }
```

Elle peut alors être utilisée partout sur une classe.

```
@MyAnnotation
public class MyClass {
    @MyAnnotation
    private int x;

    @MyAnnotation
    public void m(@MyAnnotation int y) {
        @MyAnnotation
        String m="test";
```

```
    }
}
```

4 Paramètres d'une annotation

Les paramètres d'une annotation, appelés **éléments**, déterminent les informations supplémentaires que l'on peut associer à un élément de code. Ces éléments peuvent être de différents types :

- **Types primitifs:** (int, double, boolean, etc.)
- **Types référence:** (String, objets)
- **Types de classe:** (Class)
- **Annotations:** (pour créer des annotations composées)
- **Enumérations:** (pour des valeurs prédéfinies)
- **Tableaux:** de tous les types précédents

Définition d'une annotation

```java
public @interface MyAnnotation {
    String message(); // Message associé à l'annotation
    int size() default 10; // Taille maximale (valeur par défaut : 10)
    Digit digit() default Digit.UN; // Choix entre UN et DEUX (par défaut : UN)
    Class<?>[] classes() default {}; // Tableau de classes associées

    enum Digit {UN, DEUX} // Valeurs possibles pour le chiffre
}
```

et utilisation sur une classe ou une méthode

```java
@MyAnnotation(message="hello",                    digit=MyAnnotation.Digit.UN,
classes={String.class, Arrays.class})
public class MyClass {
    @MyAnnotation(message="bye")
    public void x() {}
}
```

5 Value

Si leur seul attribut obligatoire s'appelle `value`, il est possible d'omettre son nom lors de l'annotation.

```java
public @interface MyAnnotation {
    String value();
```

```
    int size() default 10;
}
```

```
@MyAnnotation("hello")
public class MyClass { }
```

6 Portée d'une annotation

L'annotation `@Target` définit la portée d'une annotation en spécifiant les éléments de programme (classes, méthodes, champs, paramètres, etc.) auxquels elle peut être appliquée. Les valeurs possibles pour `@Target` sont : `ANNOTATION_TYPE`, `CONSTRUCTOR`, `FIELD`, `LOCAL_VARIABLE`, `METHOD`, `PACKAGE`, `MODULE`, `PARAMETER`, `TYPE`, `TYPE_PARAMETER`.

Par exemple, en utilisant `@Target(ElementType.METHOD)`, on indique que l'annotation ne peut être appliquée qu'à des méthodes.

```
//| echo: false
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;
```

```
@Target({ElementType.LOCAL_VARIABLE, ElementType.FIELD})
public @interface MyAnnotation {
    int value() default 0;
}
```

```
public class MyClass {
    @MyAnnotation
    int x;
    public void m() {@MyAnnotation int x;}
}
```

```
//| echo: true
//| output: true
//| error: true
@MyAnnotation
public class MyClass {
    int x;
}
```

```
@MyAnnotation

annotation interface not applicable to this kind of declaration
```

7 Annotations répétables 1/2

Par défaut, une annotation ne peut être appliquée qu'une seule fois à un élément.

- **Solution :** Utiliser un wrapper d'annotation pour regrouper plusieurs instances de la même annotation.

```
@Target({ElementType.LOCAL_VARIABLE, ElementType.FIELD})
public @interface MyAnnotations {
    MyAnnotation[] value();
}
```

```
public class MyClass {
    @MyAnnotations({@MyAnnotation(1),@MyAnnotation(2)})
    int x;
}
```

8 Annotations répétables 2/2

ou les autoriser à être répétables grâce à l'annotation `@Repeatable`, évitant ainsi la déclaration explicite d'un tableau d'annotations.

```
//| echo: false
import java.lang.annotation.Repeatable;
```

```
@Repeatable(MyAnnotations.class)
@Target({ElementType.LOCAL_VARIABLE, ElementType.FIELD})
public @interface MyAnnotation {
    int value() default 0;
}
```

```
public class MyClass {
    @MyAnnotation(1)
    @MyAnnotation(2)
    int x;
}
```

9 Annotations de package

Pour annoter un package, utilisez le fichier spécial `package-info.java`.

Exemple:

```
java filename="package-info.java" @SuppressWarnings("unused") package com.example;
```

10 Durée de vie et héritage des annotations

- `@Retention` Détermine à quel moment du cycle de vie d'une application une annotation est accessible.

Valeur	Description
SOURCE	Utilisée uniquement par le compilateur.
CLASS	Conservée dans le fichier .class.
RUNTIME	Conservée à l'exécution.

- `@Inherited` Indique si une annotation déclarée sur une classe est également présente sur ses sous-classes.
- `@Documented` Indique qu'une annotation doit être incluse dans la documentation générée par le javadoc.

```
//| echo: false
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.METHOD)
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
  public enum Status {IN_PROGRESS, DONE}
  Status status() default MyAnnotation.Status.DONE;
}
```

```
public class MyClass {
    @MyAnnotation
```

```
    public void m1() {System.out.println("Hello1");}

    @MyAnnotation(status=MyAnnotation.Status.IN_PROGRESS)
    public void m2() {System.out.println("Hello2");}

    @MyAnnotation(status=MyAnnotation.Status.DONE)
    public void m3() {System.out.println("Hello3");}

    public void m4() {System.out.println("Hello4");}
}
```

11 Utilisation par instrospection (1/2)

Les annotations peuvent être utilisées à l'exécution grâce à l'introspection, à condition d'avoir la rétention `RUNTIME`.

- **Class Class:** Représente une classe en mémoire.
- **Méthodes d'introspection:** `getMethod()`, `getField()`, `getConstructor()`, etc. permettent d'accéder aux membres d'une classe.
- **Annotations:** Les annotations présentes sur ces membres peuvent être récupérées.
- **Actions programmables:** Une fois qu'un élément annoté a été identifié, on peut :
  - **Invoquer une méthode:** Exécuter le code contenu dans une méthode.
  - **Accéder à une valeur:** Récupérer la valeur d'un champ.
  - **Créer une instance:** Instancier un objet à partir d'un constructeur.
  - ...

L'exemple ci-dessous affiche les méthodes de la classe précédente annotées avec `@MyAnnotation`.

```
//| echo: false
import java.lang.reflect.Method;
```

```
//| echo: true
//| output: true

// Récupère une instance de la classe MyClass
Class aClass = new MyClass().getClass();

// Récupère toutes les méthodes déclarées dans MyClass et les filtre pour ne
garder que celles annotées avec @MyAnnotation
Arrays.stream(aClass.getDeclaredMethods())
```

```
    .filter(m→m.getAnnotation(MyAnnotation.class)≠null)
                    .map(m→"La     méthode     %s     a     comme     statut
%s.".formatted(m.getName(),m.getAnnotation(MyAnnotation.class).status()))
    .forEach(System.out::println);
```

```
La méthode m2 a comme statut IN_PROGRESS.
La méthode m1 a comme statut DONE.
La méthode m3 a comme statut DONE.
```

12 Utilisation par instrospection (2/2)

Il est aussi possible d'invoquer dynamiquement ces méthodes.

```
//| echo: true
//| output: true

// Invokes methods annotated with @MyAnnotation (status=DONE) on the provided
object

MyClass myObject = new MyClass();

Arrays.stream(myObject.getClass().getDeclaredMethods())
        .filter(m → m.getAnnotation(MyAnnotation.class) ≠ null &&

m.getAnnotation(MyAnnotation.class).status().equals(MyAnnotation.Status.DONE))
        .forEach(m → {
            try {
                m.invoke(myObject);
            } catch (Exception e) {
                 System.err.println("Error invoking method " + m.getName() + ":
" + e.getMessage());
            }
        });
```

```
Hello1
Hello3
```