GRUNDLAGEN • PRAXIS • ARCHITEKTUR • AGILE • INNOVATION

1 / 2019

JAVAPRO

Magazin für professionelle Java Entwicklung in der Praxis

#JAVAPRO

www.javapro.io

www.javapro.io

#JAVAPRO #CoreJava #Annotations

Deep-Dive into Annotations – Teil 2

Java-Annotationen sind ein mächtiges Sprachmerkmal, deren Interna allgemein nicht sonderlich bekannt sind. In Teil 2 unserer dreiteiligen Serie wird aufgezeigt, wie sich eigene Annotationen erstellen und mit den notwendigen Eigenschaften versehen lassen.

Autor:

Christian Heitzmann ist Gründer und Geschäftsführer der SimplexaCode AG in Luzern, die sich auf Software-Entwicklung, -Schulung und -Beratung v.a. für MINT-Anwendungen und technische Implementierungsthemen in Java spezialisiert hat. Er ist seit 15 Jahren mit Java vertraut und hat während vieler Jahre Algorithmen und Mathematik unterrichtet.

christian.heitzmann@simplexacode.ch
https://www.simplexacode.ch

Im ersten Teil unserer Serie wurden die typischen Begegnungsszenarien zwischen Programmierer und Java aufgezeigt, nämlich die fünf Standardannotationen. Diese sind allen voran **`@Override`** und **`@SuppressWarnings("unchecked")`** sowie **`@Deprecated`**, **`@FunctionalInterface`** und **`@SafeVarargs`**, deren Hintergründe wir ausführlich beleuchtet haben. In Teil 2 geht es nun um das Implementieren eigener Annotationen und ihrer Eigenschaften. Das Erstellen eigener Annotationen ist prinzipiell denkbar einfach, wie folgende zwei Code-Beispiele zeigen:

(Listing 1)

```
public @interface MyFirstAnnotation {}
```

(Listing 2)

```
@MyFirstAnnotation
public final class MyFirstAnnotatedClass {}
```

Die Deklaration einer Annotation wird durch das Schlüsselwort **`@interface`** eingeleitet und zeigt damit auch ihre Ähnlichkeit zu Interfaces. Wie in Teil 1 bereits erwähnt, erweitert jede Annotation zwar das Interface **`java.lang.annotation.Annotation`**, umgekehrt lassen sich Annotationen aber nicht einfach dadurch erzeugen, indem man manuell diese Schnittstelle erweitert. Annotationen und Interfaces sind trotz ihrer Ähnlichkeit zwei verschiedene Paar Schuhe, die vom Compiler und im Bytecode auch so behandelt werden.

Bekanntlich steckt der Teufel im Detail: Beim Deklarieren einer eigenen Annotation sollte man nämlich drei Eigenschaften festlegen:

1. Wie lange, d. h. bis zu welcher Stufe im Entwicklungsprozess soll die Annotation erhalten bleiben?
2. Welche Zielobjekte, d. h. für welche Sprachelemente soll die Annotation möglich sein?
3. Wie viele und welche Parameter soll die Annotation anbieten?

Eigenschaft 1: Erhaltung der Annotation - @Retention

Um anzugeben wie lange eine Annotation erhalten bleiben soll, muss die eigene Annotation wiederum mit einer Meta-Annotation versehen werden, nämlich mit **`@Retention`**, übersetzbar mit "Aufbewahrung", "Beibehaltung" oder "Speicherung". Die **`@Retention`** Meta-Annotation benötigt als Parameter ein Element der Aufzählung **`java.lang.annotation.RetentionPolicy`**. Zur Auswahl stehen:

- **`RetentionPolicy.SOURCE:`** Die eigene Annotation steht nur im Quelltext zur Verfügung und wird beim Kompilieren entfernt.
- **`RetentionPolicy.CLASS:`** Die eigene Annotation steht im Bytecode der Klassendatei, aber nicht (mehr) zur Laufzeit zur Verfügung.
- **`RetentionPolicy.RUNTIME:`** Die eigene Annotation steht auch zur Laufzeit zur Verfügung, so dass mittels Reflection darauf zugegriffen werden kann.

Wird die eigene Annotation nicht mit der **`@Retention`** Meta-Annotation versehen, so gilt die Standard-Retention **`CLASS`**. Als Code-Beispiel sieht das Ganze wie folgt aus:

(Listing 3)

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.SOURCE)
@interface SourceAnnotation {}

@Retention(RetentionPolicy.CLASS)
@interface ClassAnnotation {}

@Retention(RetentionPolicy.RUNTIME)
@interface RuntimeAnnotation {}
```

Eigenschaft 2: Annotierbare Sprachelemente - @Target

In Java gibt es diverse Sprachelemente (z. B. Klassen, Methoden, Variablen etc.), aber eine Annotation macht nicht zu jedem Sprachelement Sinn. Um anzugeben, vor welchen Sprachelementen eine eigene Annotation verwendet werden darf, steht die Meta-Annotation **`@Target`** (deutsch „Ziel", „Zielobjekt") zur Verfügung. **`@Target`** erwartet als Parameter ein oder mehrere Elemente der Aufzählung **`java.lang.annotation.ElementType`**. Zur Auswahl stehen mehr oder weniger selbsterklärend:

- **`ElementType.ANNOTATION_TYPE`**
- **`ElementType.CONSTRUCTOR`**
- **`ElementType.FIELD`**
- **`ElementType.LOCAL_VARIABLE`**
- **`ElementType.METHOD`**
- **`ElementType.PACKAGE`**
- **`ElementType.PARAMETER`**
- **`ElementType.TYPE`**
- **`ElementType.TYPE_PARAMETER`**
- **`ElementType.TYPE_USE`**
- **`ElementType.MODULE`** (seit Java 9)

Wird die Meta-Annotation **`@Target`** ausgelassen, so gelten standardmäßig alle **`ElementType`** außer **`TYPE_PARAMETER`** und **`TYPE_USE`**. Folgende Code-Beispiele deklarieren zuerst eigene Annotationen – eine für jedes Target – und setzen diese dann vor die entsprechenden Sprachelemente:

(Listing 4)

```
import java.lang.annotation.*;

@Target(ElementType.ANNOTATION_TYPE)
@interface AnnotationTypeAnnotation {}

@Target(ElementType.CONSTRUCTOR)
@interface ConstructorAnnotation {}

@Target(ElementType.FIELD)
@interface FieldAnnotation {}

@Target(ElementType.LOCAL_VARIABLE)
@interface LocalVariableAnnotation {}

@Target(ElementType.METHOD)
@interface MethodAnnotation {}

@Target(ElementType.PACKAGE)
@interface PackageAnnotation {}
```

```
@Target(ElementType.PARAMETER)
@interface ParameterAnnotation {}

@Target(ElementType.TYPE)
@interface TypeAnnotation {}

@Target(ElementType.TYPE_PARAMETER)
@interface TypeParameterAnnotation {}

@Target(ElementType.TYPE_USE)
@interface TypeUseAnnotation {}
```

(Listing 5)

```
@TypeAnnotation
@TypeUseAnnotation
public final class TargetAnnotationsDemo
    <@TypeParameterAnnotation @TypeUseAnnotation T> {

  @TypeAnnotation
  @TypeUseAnnotation
  private static enum GreekLetter {
    @FieldAnnotation ALPHA,
    @FieldAnnotation BETA,
    @FieldAnnotation GAMMA
  }

  @FieldAnnotation
  @TypeUseAnnotation
  private GreekLetter greekLetterField;

  @ConstructorAnnotation
  @TypeUseAnnotation
  public TargetAnnotationsDemo() {
    greekLetterField = GreekLetter.ALPHA;
  }

  @MethodAnnotation
  public static void main
    (@ParameterAnnotation @TypeUseAnnotation String args[]) {

    @LocalVariableAnnotation
    @TypeUseAnnotation
    TargetAnnotationsDemo<@TypeUseAnnotation String> demo
        = new @TypeUseAnnotation TargetAnnotationsDemo<>();

    System.out.println("Value of Greek letter field: "
                       + demo.greekLetterField);
  }
}
```

(Listing 6)

```
/**
 * The package-info.java file usually contains the package
 * documentation in Javadoc format.
 */
@PackageAnnotation
package ch.simplexacode.annotations;
```

(Listing 7)

```
@AnnotationTypeAnnotation
@TypeAnnotation
@TypeUseAnnotation
@interface AnnotatedAnnotation {}
```

Anmerkungen:

- Die Annotation-Targets **`CONSTRUCTOR`**, **`FIELD`**, **`LOCAL_VARIABLE`**, **`METHOD`**, **`PARAMETER`** und **`TYPE_PARAMETER`** sollten anhand der Code-Beispiele selbsterklärend sein.
- Die Annotation mit dem Parameter **`TYPE`** gilt für alle Typdeklarationen, d. h. für Klassen, Interfaces, Enumerations und wiederum Annotationen. Ein Target, welches z. B. ausschließlich für Klassen gilt, ist nicht möglich.
- Pakete dürfen nur in der zum Paket gehörenden Datei **`package-info.java`** annotiert werden, dort wo sich im Idealfall auch die Paketdokumentation befindet. Denn bei mehreren Klassen in einem Paket müsste entweder konsequent jedes **`package`** Statement mit einer Annotation des Targets **`PACKAGE`** annotiert werden, oder es käme zu Widersprüchen.
- Annotationen können auch selbst mit eigenen Annotationen versehen werden. Prinzipiell lässt sich jede Annotation mit anderen (Meta-)Annotationen der Targets **`ANNOTATION_TYPE`**, **`TYPE`** und **`TYPE_USE`** annotieren.
- Jede Verwendung eines Typs kann auch mit einer Annotation des Targets **`TYPE_USE`** versehen werden. Dies ist auch der Grund, warum im Code-Beispiel fast überall, wo eine Annotation angebracht werden kann, auch eine solche mit Target **`TYPE_USE`** gültig ist. Ausnahmen im Code-Beispiel sind die **`enum`** Elemente und die Methode.
- Selbstverständlich lassen sich auch mehrere Targets angeben. Dies sieht dann z. B. wie folgt aus:

(Listing 8)

```
@Target({
  ElementType.TYPE, ElementType.FIELD, ElementType.METHOD
})
@interface SeveralTargetsAnnotation {}
```

Eigenschaft 3: Parameter der Annotation

Dem aufmerksamen Leser wird nicht entgangen sein, dass einige Annotationen für sich alleine stehen, z. B. **`@Override`**, andere hingegen mit Parametern versehen werden, z. B. **`@SuppressWarnings("unchecked")`** oder **`@Retention(RetentionPolicy.RUNTIME)`**. Auch für eigene Annotationen lässt sich angeben, ob und welche Parameter sie annehmen sollen. Die Definition der Annotationselemente lässt sich erneut am besten anhand von Code-Beispielen zeigen:

(Listing 9)

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.CLASS)
```

```
@Target(ElementType.TYPE)
@interface Author {
  public static enum Country {
    GERMANY, AUSTRIA, SWITZERLAND
  }

  public String firstName();
  public String lastName();
  public Country country() default Country.GERMANY;
  public boolean selfEmployed() default true;
}

@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.CONSTRUCTOR, ElementType.METHOD })
@interface ErrorCodes {
  public int[] value();
}

@Retention(RetentionPolicy.SOURCE)
@Target({ ElementType.FIELD, ElementType.LOCAL_VARIABLE })
@interface Initialize {
  public String initializationString();
}

@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.FIELD, ElementType.LOCAL_VARIABLE })
@interface Serialize {
  public boolean encrypt() default false;

}
```

(Listing 10)

```
@Author(firstName = "Christian",
        lastName = "Heitzmann",
        country = Author.Country.SWITZERLAND)
        /* selfEmployed = true */
public final class AttributedAnnotationsDemo {

  @Initialize(initializationString = "foo")
  @Serialize  /* encrypt = false */
  private int field1;

  @Initialize(initializationString = "bar")
  @Serialize(encrypt = true)
  private String field2;

  @ErrorCodes(42)
  public AttributedAnnotationsDemo() {}

  @ErrorCodes(value = 1764)
  public void method1() {}

  @ErrorCodes({ 1, 23, 456, 7890 })
  public void method2() {}
}
```

Anmerkungen:

- Prinzipiell lassen sich Annotationen mit Schlüssel-Wert-Paaren der Form **`Schlüssel = Wert`** parametrisieren. Den Namen des Schlüssels und den Typ implementiert man im Annotations-"Interface" im Stil normaler Schnittstellenmethoden. Der Name der Methode stellt den Schlüssel dar, der "Rückgabetyp" den Typ des Wertes. Prinzipiell kann auf das Schlüsselwort **`public`** wie bei allen Interfaces verzichtet werden, da alle Methoden einer Schnittstelle implizit **`public`** sind. Gleich verhält es sich bei Annotationen, die den Interfaces ja ähnlich sind. Es ist eine persönliche Stilfrage, ob bei Schnittstellen wie auch bei eigenen Annotationen trotzdem konsequent das Schlüsselwort **`public`** mit aufgeführt wird.
- Im Code-Beispiel wird eine eigene Annotation **`@Author`** definiert, die vier Schlüssel-Wert-Paare enthält. **`firstName`** und **`lastName`** sind vom Typ **`String`** und werden in der Demo-Klasse auch entsprechend gesetzt: **`firstName = "Christian"`**, **`lastName = "Heitzmann"`**. Als **`country`** lässt sich ein Element des Aufzählungstyps **`Country`** angeben, der direkt innerhalb von **`@Author`** definiert wurde.
- Die Elemente **`country`** und **`selfEmployed`** können weggelassen werden. Durch das Schlüsselwort **`default`** wurden für sie Standardwerte festgelegt. Im obigen Beispiel wurde das Land des Autors mit **`SWITZERLAND`** überschrieben (sonst wäre es standardmässig **`GERMANY`** gewesen), die Eigenschaft **`selfEmployed`** wurde hingegen beim Standard **`true`** belassen.
- Im Code-Beispiel lassen sich den Konstruktoren und Methoden je beliebig viele **`ErrorCode`** in Form von Integers mitgeben. In der Deklaration ist dies am Array **`int[]`** zu erkennen. In der Anwendung können ein einzelner Wert, z. B. **`@ErrorCodes(42)`**, oder mehrere Werte, z. B. **`@ErrorCodes({ 1, 23, 456, 7890 })`**, in geschweiften Klammern übergeben werden.
- Hat ein Annotationstyp nur einen einzigen Schlüssel, und hat dieser den Namen **`value`**, so kann bei der Übergabe des Parameters auf seine Nennung verzichtet werden, so wie im Konstruktor und in der Methode **`method2`** gezeigt. In **`method1`** wird **`value`** explizit aufgeführt, obwohl es nicht nötig wäre.
- In den Definitionen der eigenen Annotationstypen **`@Initialize`** und **`@Serialize`** lauten die Schlüsselnamen nicht **`value`**, sondern anders. In beiden Fällen muss der Schlüsselname daher immer angegeben werden.

@Repeatable

Seit Java 8 gibt es die Meta-Annotation **`@Repeatable`**, die es erlaubt eigene Annotationen mehrfach hintereinander, also wiederholt einzusetzen. Dies soll in folgenden Code-Beispielen anhand von **`ErrorCode`** gezeigt werden. Im vorangegangenen Beispiel haben wir gesehen, dass sich mehrere **`ErrorCode`** in geschweiften Klammern mitgeben lassen. Mit **`@Repeatable`** ist es auch möglich, dies anhand mehrerer einzelner Befehle zu tun. Die Definition von **`@Repeatable`** Annotationen ist etwas sperrig. Zur klareren Kennzeichnung wurde **`@ErrorCode`** in **`@SingleErrorCode`** umbenannt, denn es benötigt neu auch einen **`@ErrorCodeContainer`**, der die einzelnen **`@SingleErrorCode`** aufnimmt.

(Listing 11)

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Repeatable(ErrorCodeContainer.class)
@interface SingleErrorCode {
  public int value();

}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface ErrorCodeContainer {
  public SingleErrorCode[] value();
}
```

(Listing 12)

```
public final class RepeatableAnnotationsDemo {

  @SingleErrorCode(42)
  public void method1() {}

  @SingleErrorCode(1)
  @SingleErrorCode(23)
  @SingleErrorCode(456)
  @SingleErrorCode(7890)
  public void method2() {}
}
```

Für die etwas komplizierte Definition wird man jedoch im Code der Anwendung mit einer eleganteren Syntax entschädigt.

Weitere Meta-Annotationen: @Inherited und @Documentend

Wird eine Oberklasse annotiert, so gilt diese Annotation nicht (mehr) für ihre Unterklassen. Sollen eigene Annotationen dennoch an Unterklassen weitervererbt werden, so muss man sie mit der Meta-Annotation **`@Inherited`** versehen. Diese Vererbungsregeln gelten aber ausschließlich für Klassen, nicht für Interfaces. Annotationsvererbung bei Schnittstellen ist nicht möglich. Ein Code-Beispiel dazu folgt in der nächsten JAVAPRO - Ausgabe im 3. Teil unserer Serie, in der es um programmgesteuerte Abfragen von eigenen Annotationen geht. Standardmäßig taucht eine Annotierung (also die Anwendung einer Annotation, nicht deren Definition) in generierten Javadocs nicht auf. Je nach Zweck der Annotation kann es aber wünschenswert sein, in Javadoc ausdrücklich auf eine angebrachte Annotation oder ihre Werte hinzuweisen. Soll also eine eigene Annotation in Javadocs erscheinen, so sind diese mit einer **`@Documented`** Meta-Annotation zu versehen.

Fazit

Die Syntax für die Definition eigener Annotationstypen mit all ihren Eigenschaften ist teilweise etwas gewöhnungsbedürftig. Die Hürde für den Einsatz eigener Annotationen wäre sicher niedriger, wenn die Sprachdesigner mehr auf eine einfache Syntax geachtet hätten. Im dritten und letzten Teil unserer Serie geht es darum, wie Annotationen programmatisch abgefragt und verwendet werden können.