Die JAVAPRO ist kostenlos!
Einfach anfordern unter: www.java-pro.de

12 NEUES IN JAVA 11

33 SOFTWARE-MODERNISIERUNG MIT MICROSERVICES UND CONTAINER

GRUNDLAGEN • PRAXIS • ARCHITEKTUR • AGILE • INNOVATION

3 / 2018

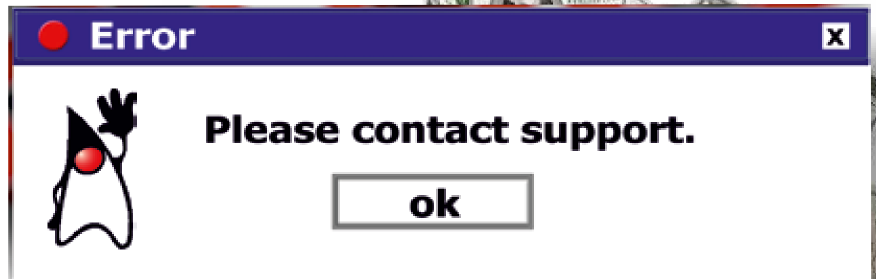
JAVAPRO

Magazin für professionelle Java Entwicklung in der Praxis

#JAVAPRO

Oracle ändert Lizenz- & Support Modell !

Java jetzt kostenpflichtig?



10 JAVA 11 - DIE FREIE JDK-WAHL

16 DEEP-DIVE INTO ANNOTATIONS - TEIL 1

24 APPLICATION-SERVER, CONTAINER ODER LIEBER GLEICH SERVERLESS?

39 NEUN BEST-PRACTICES FÜR CONTAINER

46 IDE-WARS - DIE FOUNDATION SCHLÄGT ZURÜCK

52 AGILE TRANSITION BRAUCHT KULTURELLEN WANDEL

#JAVAPRO #Annotation

Deep-Dive into Annotations - Teil 1

Java-Annotationen sind ein mächtiges Sprachmerkmal, deren Interna vielen Entwicklern wahrscheinlich nicht sehr bekannt sind. In Teil 1 unserer dreiteiligen Serie wird aufgezeigt, was Annotationen eigentlich sind, und für welche Szenarien die fünf Standard-Annotationen gedacht sind.

Mit Version 5 der Java-Plattform, Standard-Edition (Java SE) im Jahre 2004 erhielten Annotationen nebst diversen anderen bedeutenden Sprachänderungen erstmals Einzug in die mächtige Programmiersprache. Obwohl deren Einführung also bereits 14 Jahre her ist, führen Annotationen gefühlt heute noch immer ein Schattendasein. In der Populärliteratur zur Java SE werden meist nur die gängigen Standardannotationen wie z. B. **`@Override`** oder **`@SuppressWarnings ("unchecked")`** angesprochen. Beschreibungen zu komplizierteren Annotationen wie z. B. **`@SafeVarargs`** findet man sogar in der einschlägigen Spezialliteratur nur selten.

Jeder Java-Programmierer wird mit Annotationen sicher schon seine Bekanntschaft gemacht haben. Sei es mehr oder weniger freiwillig, dass einen z. B. die Verwendung des soeben erwähnten **`@Override`** überzeugt hat bzw. ans Herz gelegt wurde, oder dass man mit einem **`@SuppressWarnings("unchecked")`** am richtigen Ort im Quelltext endlich seine Ruhe vor den lästigen Compiler-Warnungen hat, die auch nach langem Tüfteln mit Generics, Type-Casts und Arrays nicht auf dem regulären Weg verschwinden wollten.

Die Beschränkung auf die zwei typischen Standardannotationen verfehlt allerdings die Tatsache, dass es sich bei Annotationen um ein sehr flexibles und mächtiges Sprachmerkmal handelt, dessen Potenzial es noch auszuschöpfen gilt. In Teil 1 dieser Serie werden zuerst die bekannten wie auch selteneren Standardannotationen vorgestellt. In Teil 2 wird anschliessend

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

aufgezeigt, wie sich eigene Annotationstypen erstellen lassen und in Teil 3, wie man diese Annotationen programmgesteuert auswerten kann. Denn gerade in beiden letzteren liegt die eigentliche Stärke der Annotationen, mit dem Java die Tür hin zur deklarativen Programmierung einen Spalt weit geöffnet hat.

Definition und Beispiele

Die wohl prägnanteste Definition für Annotationen findet sich gleich im ersten Satz von Abschnitt „9.7. Annotations" der Java Language Specification: „An annotation is a marker which associates information with a program construct, but has no effect at run time."

Bei Annotationen handelt es sich also um Metainformationen, oder salopp ausgedrückt, um „Informationen über Informationen", vergleichbar mit der ISBN-Nummer eines Buches. Die ISBN-Nummer ändert nichts am Inhalt des Buches (den eigentlichen Informationen), denn die Inhalte des Buches lassen sich genau gleich erschließen, unabhängig davon, ob das Buch nun eine ISBN-Nummer trägt oder nicht.

Annotationen tun selbst nichts, sie deklarieren nur Programmkonstrukte. Ein Buch mit einer ISBN-Nummer kann sich nicht selbst zum Haushalt eines Kunden liefern. Dazu braucht es immer noch jemand anderen, der das Buch entweder manuell oder automatisiert verpackt, transportiert und ausliefert. Gleich verhält es sich bei Annotationen. Um mit ihnen etwas anfangen zu können, braucht es einen Dritten, der den Java-Quelltext oder die Klassendatei inspiziert und bei Annotationen entsprechend reagiert, z. B.:

- Der Compiler, der die Korrektheit der Standardannotationen (z. B. **@Override**) im Zusammenspiel mit dem Quelltext überprüft und bei Bedarf eine Fehlermeldung generiert.
- Ein Code-Analyse-Tool, welches die technische Code-Qualität beurteilt, dabei aber bewusst platzierte Ausnahmebemerkungen (also Annotationen) des Entwicklers im Quelltext in der Analyse ignoriert.
- Ein Test-Framework, in denen die Testfälle annotiert werden, so dass sie als automatisierte Tests auffindbar, korrekt ausführbar und sinnvoll auswertbar sind.
- Ein Web-Server, der einen Java-Webservice, seine Methoden und deren Parameter anhand von Annotationen erkennt, automatisch konfiguriert und korrekt typisiert.
- Eine Java-XML-API, die entsprechend annotierte Java-Komponenten automatisch und korrekt zwischen Java und XML abbilden kann.
- Eigene Programme, die andere Klassen oder Objekte inspizieren, insbesondere mit Hilfe von Reflection oder sogenannten Annotationsprozessoren.

Die Grundidee von Annotationen war eigentlich auch in Java 5 nicht mehr neu. Bereits Javadoc, welches seit den Urtagen von Java existiert, ermöglichte dem Programmierer in seinen Kommentaren zu den Paket-, Klassen- und Methodenbeschreibungen gewisse Javadoc-Tags wie z. B. **@author**, **@param**, **@return** oder **@throws** zu verwenden, die dann vom Javadoc-Tool ausgewertet wurden und zu hervorragend strukturierten und indexierten HTML-Dokumentationen führten. Während die Javadoc-Kommentare und die daraus resultierenden HTML-Dokumentationen nur für Menschen als Zielpublikum gedacht sind, sind die Java-Annotationen vor allem für Maschinen gemacht. Einfach ausgedrückt, kann man Annotationen auch als Javadoc für Maschinen verstehen. Dass sowohl den Javadoc-Tags als auch den Annotationen ein **@** vorangestellt wird, ist also sicher kein Zufall.

Standard-Annotationen

Ein Blick in die Liste der All-Known-Implementing-Classes der Dokumentation des Interfaces **java.lang.annotation.Annotation** verrät, welche und wie viele vordefinierte Annotationen es in der Java SE gibt. Jede Annotation erweitert die Schnittstelle **Annotation**, allerdings lassen sich keine eigenen Annotationen erzeugen, indem man einfach dieses Interface erweitert (mehr dazu in Teil 2). Für Java 8 zählt man über 80 Annotationen. Ähnlich wie bei den Exceptions befinden sich dabei aber sehr viele in Spezialpaketen (beginnend mit **javax**) oder dienen nur ganz speziellen und eher untypischen Anwendungszwecken. Als Standardannotationen im engeren Sinn gibt es in der Java SE fünf im Paket **java.lang** (namentlich **@Deprecated**, **@FunctionalInterface**, **@Override**, **@SafeVarargs** und **@SuppressWarnings**) sowie sechs im Paket **java.lang.annotation**, wobei letztere lediglich Annotationen für Annotationen darstellen, sich also selbst annotieren. In Teil 2 werden die sogenannten Metaannotationen vorgestellt, nämlich die beiden wichtigen Annotationen **@Retention** und **@Target** sowie **@Documented**, **@Inherited**, **@Native** und **@Repeatable**. Im folgenden Code-Beispiel, das lediglich zur Erklärung konstruiert wurde und nicht für den Einsatz in der Praxis gedacht ist, werden alle fünf Standardannotationen verwendet.

(Listing 1)

```
import java.util.*;

public final class StandardAnnotationsDemoClass
    implements StandardAnnotationsDemoInterface {

  public StandardAnnotationsDemoClass() {}

  public static void main(String[] args) {
    StandardAnnotationsDemoClass demoClass
        = new StandardAnnotationsDemoClass();
    StandardAnnotationsDemoInterface.staticInterfaceMethod();
    demoClass.defaultInterfaceMethod();
    demoClass.regularInterfaceMethod();
    demoClass.safeVarargsMethod(Optional.of("Alpha"),
                                Optional.of("Beta"),
                                Optional.of("Gamma"));
    demoClass.outdatedMethod();
  }
```

```
  @Override
  public void regularInterfaceMethod() {

  /* Fails at compile time. Note the typo! */
//  public void regluarInterfaceMethod() {

    System.out.println("I'm a regular interface method.");
  }

  /* Emits compiler warnings if annotation removed. */
  @SafeVarargs
  public final void safeVarargsMethod(Optional<String>...
                                    arguments) {
    Object[] objectArray = arguments;
    objectArray[1] = Optional.of("Delta");

    /* Pollutes heap and makes loop below fail. */
//    objectArray[1] = Optional.of(new Integer(1));

    /* Emits compiler warning if annotation removed. */
    @SuppressWarnings("unchecked")
    Optional<String>[] stringOptionalArray
        = (Optional<String>[]) objectArray;

    for (Optional<String> currentStringOptional
         : stringOptionalArray) {
      String string = currentStringOptional.get();
      System.out.println(string);
    }
  }

  @Deprecated
  public void outdatedMethod() {
    System.out.println("I'm an outdated method.");
  }
}

@FunctionalInterface
interface StandardAnnotationsDemoInterface {

  public static void staticInterfaceMethod() {
    System.out.println("I'm a static interface method.");
  }

  public default void defaultInterfaceMethod() {
    System.out.println("I'm a default interface method.");
  }

  public void regularInterfaceMethod();

  /* Fails at compile time. */
//  public void anotherRegularInterfaceMethod();
}
```

Im Folgenden werden die einzelnen Standard-Annotationen in der Reihenfolge aufsteigender Komplexität kurz erläutert.

@Deprecated

Veraltete Methoden, die nicht mehr verwendet werden sollen, werden mit der Annotation **`@Deprecated`** versehen. Dies entspricht dem Javadoc-Tag **`@deprecated`** (klein geschrieben!), nur dass mit der Annotation diese Information nun auch dem Compiler zur Verfügung steht. In Eclipse wird im Code-Beispiel die **`outdatedMethod`** durchgestrichen dargestellt. Denkbar wäre auch eine Compiler-Warnung. Im Gegensatz zum Javadoc-Tag ist die Annotation immer verfügbar, also auch dann, wenn der Java-Quelltext oder die Dokumentation einer Bibliothek mit einer als deprecated-markierten Methode nicht zur Verfügung steht, oder wenn in der IDE die Verknüpfung zur Dokumentation nicht richtig eingerichtet ist.

@Override

Die **`StandardAnnotationsDemoClass`** implementiert das **`StandardAnnotationsDemoInterface`**, deren **`regularInterfaceMethod`** es zu überschreiben gilt. Wird eine Methode einer Oberklasse oder eines Interfaces überschrieben, so kann der Programmierer dies mit einer **`@Override`** Annotation versehen, was jedoch kein Muss ist. Dennoch ist es empfehlenswert, in der IDE den Compiler so einzustellen, dass bei fehlender Annotation eine Warnung ausgegeben wird.

Der Zweck ist folgender: Wird bei einer im guten Glauben überschreibenden Methode die **`@Override`** Annotation angebracht, dann MUSS die damit annotierte Methode auch wirklich überschreiben. Wenn die zu überschreibende Methode in der Oberklasse oder im Interface nicht (mehr) existiert, oder wenn die Methodensignaturen nicht identisch sind, was insbesondere bei Schreibfehlern oder falschen Parametertypen passieren kann, führt dies zu einem Compiler-Fehler. Das kann man ausprobieren, indem an die Methode **`regularInterfaceMethod`** durch **`regLUarInterfaceMethod`** ersetzt, die mit einem Schreibfehler behaftet ist. **`@Override`** stellt somit eine Absicherung dar, um in der Praxis schwierig auffindbare Fehler durch fehlerhaftes Überschreiben von Methoden zu vermeiden. Die Folgen eines kleinen Schreibfehlers können nämlich verheerend sein, insbesondere bei nicht abstrakten Oberklassen: Der Compiler wird sich nicht beschweren, die Laufzeitumgebung ebenso wenig, aber der Polymorphismus geht verloren. Das Programm wird stillschweigend die falsche Methode ausführen, nämlich diejenige der Oberklasse.

@FunctionalInterface

Die Annotation **`@FunctionalInterface`** kam mit Java 8 hinzu, was mit der elementaren Bedeutung der Functional-Interfaces in Zusammenhang mit der ebenfalls in Java 8 eingeführten funktionalen Programmierung und den Lambda-Ausdrücken zu erklären ist. Zur Erinnerung: Eine funktionale Schnittstelle ist ein Interface, das genau eine abstrakte Methode besitzt, mit Ausnahme der Methoden aus **`Object`**. Seit Java 8 lassen sich Interfaces ebenfalls mit Default-Implementierungen und statischen Implementierungen versehen. Diese Methoden sind ausdrücklich nicht abstrakt und können daher in beliebiger Menge auch in einer funktionalen Schnittstelle vorkommen. So ist z. B. **`java.util.Comparator<T>`** eine funktionale Schnittstelle, obwohl sie aus 18 Methoden besteht.

Die Annotation **@FunctionalInterface** markiert ein Interface als funktionale Schnittstelle. Ähnlich wie die Annotation **@Override** dient auch sie damit als Sicherheit, falls die Schnittstelle eines Tages durch Hinzufügen einer weiteren abstrakten Methode plötzlich nicht mehr funktional sein sollte. In der Folge würden nämlich sämtliche Lambda-Ausdrücke nicht mehr funktionieren. Würde man in aufgezeigtem Code-Beispiel der funktionalen Schnittstelle **StandardAnnotationsDemoInterface** die zusätzliche abstrakte Methode **anotherRegularInterfaceMethod** hinzufügen, so würde dies aufgrund der **@FunctionalInterface** Annotation sofort zu einem Compiler-Fehler führen.

@SuppressWarnings

In der Methode **safeVarargsMethod** des Code-Beispiels wird ein **objectArray** (also ein Array vom Typ **Object[]**) einem Array **stringOptionalArray** vom generischen Typ **Optional<String>[]** zugewiesen. Es ist klar, dass dieses „maskierte" **Object**-Array zur Laufzeit ein **Optional** Array sein muss, da es ansonsten sofort eine unmissverständliche **ClassCastException** geben würde. Weniger klar hingegen ist, welchen generischen Typ die **Optionals** im Array haben, denn Java verfügt bei der Verwendung von Generics im Zusammenspiel mit Arrays diesbezüglich über keine Laufzeitinformationen. Auch hat der Compiler keine Möglichkeit, die Typsicherheit vorab sicherzustellen. Er gibt daher eine Warnung aus: **Type safety: Unchecked cast from Object[] to Optional<String>[]**. Mit der Annotation **@SuppressWarnings("unchecked")** gibt der Programmierer dem Compiler ein Versprechen ab, sich von der Typsicherheit vergewissert zu haben, also dass es sich in diesem Array wirklich um **Optional<String>** und nicht etwa um **Optional<Integer>** handelt. Der Compiler kann daraufhin auf seine Warnung verzichten.

Es versteht sich von selbst, dass eine **@SuppressWarnings** Annotation nicht leichtfertig angewendet werden darf und nur den kleinstmöglichen Geltungsbereich umfassen soll. Man sollte also auf keinen Fall eine ganze Methode oder gar eine ganze Klasse damit annotieren! Das Zusammenspiel von Generics und Arrays in Java ist speziell und alles andere als trivial.[1]

@SafeVarargs

Seit Java 5 kann man Methoden eine variable Anzahl an Parametern übergeben. Deklariert werden solche Parameter mit Auslassungspunkten, z. B. **safeVarargsMethod(Optional<String>... arguments)** in vorliegendem Code-Beispiel. In Java bezeichnet man diese variable Anzahl an Methodenparametern oder -argumenten als Varargs. Intern werden Varargs ganz einfach durch ein Array entsprechenden Typs und entsprechender Größe realisiert. Jedes Argument des Varargs entspricht dann einem Element des Arrays, auf das die Methode Zugriff hat. Bei **Optional<String>...** führt diese Umwandlung zu einem generischen Array **Optional<String>[]**. Wie im vorangegangenen Punkt zu **@SuppressWarnings** bereits erwähnt, vertragen sich Generics und Arrays nicht sonderlich gut, was der Compiler direkt, wenn auch unverständlich, zur Sprache bringt. Beim Aufrufer meldet er: **Type safety: A generic array of Optional<String> is created for a varargs parameter.** Bei der Methodendeklaration lautet die Warnung: **Type safety: Potential heap pollution via varargs parameter arguments**. Diese Heap-Verschmutzung kann insbesondere dann auftreten, wenn das Array anschliessend ungeschickt beschrieben oder aus der Hand gegeben wird, also die Methode verlässt. Ein nicht vertrauenswürdiger Client (und davon ist grundsätzlich immer auszugehen) könnte dieses Array dann direkt bearbeiten.

Im Code-Beispiel ist eine solche Möglichkeit einer Heap-Pollution aufgezeigt: Das Argumenten-Array **arguments** vom Typ **Optional<String>[]** wird einem **Object[]** zugewiesen. In diesem **Object**-Array können die Elemente auch beschrieben werden. Im Code-Beispiel wird das Überschreiben des Elementes am Index **1** von **Optional.of("Beta")** auf **Optional.of("Delta")** funktionieren. Erstaunlicherweise wird aber auch die (noch auskommentierte) Zuweisung mit **Optional.of(new Integer(1))** funktionieren, ohne dass es an dieser Stelle zu einer Exception kommt. Erst weiter unten im Code, wenn in der Schleife das Element aus dem **Optional** extrahiert und auf einen **String** gecastet wird, kommt es zu einer **ClassCastException**. Auf diesen Umstand möchte der Compiler mit seiner Warnung hinweisen.

Hat sich der Programmierer davon vergewissert, dass eine derartige Heap-Pollution nicht auftreten kann, so kann er die Warnung mit einer **@SafeVarargs** Annotation deaktivieren. Diese Garantie besteht auf jeden Fall dann, wenn die Methode das Argumenten-Array weder beschreibt noch von außen Zugriff auf dieses Array gewährt – auch nicht aus Versehen! Für den normalen, klassischen Anwendungsfall von Varargs dürfte es eigentlich keine Schwierigkeiten geben.

Fazit:

Die fünf hier vorgestellten Standardannotationen dienen (noch) ausschließlich dem Compeiler. Außer einer gewissen Sicherheit und allenfalls etwas weniger händischer Dokumentationsarbeit ist der Vorteil von Annotationen noch nicht auf Anhieb offensichtlich. Aus diesem Grund werden in den folgenden beiden Teilen die Definition und Auswertung eigener Annotationen detailliert vorgestellt.

Quellen:

1 Mehr zum Thema: link.simplexacode.ch/rk57