Annotationen in Java 1.5

Piotr Lorenz (wi5874)

22. Dezember 2005

Inhaltsverzeichnis

1 Einleitung

Annotationen sind eine Neuerung in J2SE 1.5 und werden häufig auch als Metadaten, also Anmerkungen zum Quellcode bezeichnet. Im Allgemeinen bezeichnet man Daten als Metadaten, die mit dem eigentlichen Gegenstand der Betrachtung zwar inhaltlich in Verbindung gebracht werden können, aber deren Vorhandensein den Inhalt des betrachteten Gegenstandes nicht verändern. Bei der Betrachtung eines Buches könnte man den Namen des Autors, den Verlag oder die ISBN als Metadaten bezeichnen. Stünden diese Daten nicht zur Verfügung, so würde das nichts am Inhalt des Buches ändern. Wollte man allerdings ein Buch bestellen, wären diese Daten unerlässlich und somit keine Metadaten mehr. Es kommt also darauf an, unter welchem Gesichtspunkt man die Daten betrachtet. Eine klare Trennung von Metadaten und normalen Daten ist somit nicht möglich.

Bislang konnte man dem Java-Quellcode Metadaten in Form von `Doclets` innerhalb von JavaDoc-Kommentaren hinzuzufügen. Diese wurden dann von Tools wie z.B. `XDoclet` ausgewertet und verarbeitet. Zum Teil wurden mithilfe dieser Tools externe Dateien generiert, welche dann zu Laufzeit geparst und ausgewertet wurden. Mitunter wurde auch neuer Quellcode generiert. Annotationen ermöglichen es nun, in einem Java-Quellcode zusätzliche Informationen in einheitlicher Form einzubringen, welche später durch ein zusätzliches Tool ausgewertet werden können.[1] Auch der Compiler ist ein Tool welches Quellcode analysiert und verarbeitet. Es gibt z.B. spezielle Annotationen die bewirken, dass bestimmte Warnungen des Compilers auszugeben bzw. unterdrücken werden. Andere Annotationen werden vom Compiler in die .class Datei eingebunden, so dass sie später von der VM [2] oder anderen Tools ausgewertet werden können.

[1] vergl. [Now04] Seite 165
[2] Virtual Machine

2 Syntax

Dieses Kapitel behandelt die Syntax von Annotationen.

Sollen Annotationen benutzen werden, so muß erst ein Typ zu dieser Annotation deklariert werden. Annotations-Typen werden durch ein Annotations-Interface definiert, wobei es sich hierbei um eine Sonderform der `interface` Deklaration handelt:

```
AnnotationTypeDeclaration:
        @ interface Identifier AnnotationTypeBody

    AnnotationTypeBody:
        { [AnnotationTypeElementDeclarations] }

    AnnotationTypeElementDeclarations:
        AnnotationTypeElementDeclaration
        AnnotationTypeElementDeclarations AnnotationTypeElementDeclaration

AnnotationTypeElementDeclaration:
        {Modifier} AnnotationTypeElementRest

AnnotationTypeElementRest:
         Type Identifier AnnotationMethodOrConstantRest;
        ClassDeclaration
        InterfaceDeclaration
        EnumDeclaration
        AnnotationTypeDeclaration

        AnnotationMethodOrConstantRest:
        AnnotationMethodRest
        AnnotationConstantRest

AnnotationMethodRest:
        ( ) [DefaultValue]

AnnotationConstantRest:
        VariableDeclarators

    DefaultValue:
        default ElementValue
```

[3]

[3] vergl. [GJSB05] Kapitel 18

Eine konkrete Deklaration eines solchen Annotations-Typen könnte demnach wie folgt aussehen:

```
1 public @interface MyFirstAnnotation {
2   boolean notNull() default false;
3   int someNumber() default 4711;
4   String column();
5 }
```

Die Anwendung eines solchen Annotations-Typen, also die Annotation an sich hat immer das folgende Format:

```
Annotation:
        @ TypeName [( [Identifier =] ElementValue)]
```

3

Eine konkrete Anwendung könnte demnach wie folgt aussehen:

```
1 //Wenn man alle Parameter setzen will
2 @MyFirstAnnotation(notNull=false,someNumber=12,column="HelloWorld")
3 //...
4
5 //Wenn man die Standard Werte nutzen will
6 @MyFirstAnnotation
7 //...
```

Als Parameter einer Annotation sind folgende Typen zulässig:

- primitive Typen
- `String`
- `Class` (optional `Class<? extends OtherClass>`)
- Aufzählungs-Typen (`enum`)
- Annotations-Typen
- eindimensionale Arrays der vorangegangenen Typen

Annotationen können durch die Anzahl ihrer Parameter in drei Gruppen geteilt werden, für die es folgende Kurzschreibweisen gibt:

Parameter	**Bezeichnung**	**Kurzform**
$= 0$	Marker-Annotationen	`@SomeAnnotation`
$= 1$	Einzelwert-Annotationen	`@SomeAnnotation("value")`
> 1	mehrwertige Annotationen	`@SomeAnnotation(key1="value",key2=4711,...)`

Bei der verkürzten Schreibweise für Einzelwert-Annotationen ist zu beachten, dass diese Schreibweise nur dann zulässig ist, wenn der Parameter den Namen `value` hat. Sowohl der `default` Wert, als auch die Werte anderer Parameter dürfen nie auf `null` gesetzt sein und müssen zum Compile-Zeitpunkt feststehen.[4]

Annotationen sind *Modifikatoren*:

```
Modifier:
                Annotation
                public
                protected
                private
                static
                abstract
                final
                native
                synchronized
                transient
                volatile
                strictfp
```

3

Sowohl Pakete als auch Klassen, Schnittstellen, Felder, Methoden, Parameter, Konstruktoren oder lokale Variablen können mit Annotationen versehen werden (Siehe [GJSB05] Kapitel 18). Wobei es für einzelnen Annotationen Einschränkungen geben kann. Es kann also sein, dass eine bestimmte Annotation laut Definition nur auf Pakete oder Methoden anzuwenden ist. Wie man die Ziele (*Targets*) von Annotationen bestimmt, beschreibt das Kapitel *Meta-Annotationen*.

[4] vergl. [HC05] Seite 1151

3 Standard Annotationen

Im Paket `java.lang` des J2SDK 1.5 sind standard Annotationen enthalten, die sinnvoll eingesetzt sehr nützlich sein können. Dieses Kapitel befasst sich mit der Funktionsweise und der Anwendung eben dieser Annotationen. vergl. [HC05] Seite 1153

Annotation	**Anwendbar auf**	**Wirkung**
Deprecated	alles	Markiert das Element als veraltet.
Override	Methoden	Gibt an, dass diese Methode eine Superklassenmethode überschreibt.
SupressWarnings	alles, außer Pakete und Annotationen	Unterdrückt Warnungen des gegebenen Typs.

@Deprecated

`@Deprecated` ist eine Annotation, welche ohne Einschränkung auf alle annotierbaren Elemente anwendbar ist, auch wenn sie bei einer Anwendung auf lokale Variablen oder Parameter keine Wirkung zeigt. Werden Elemente verwendet, die mit dieser Annotation gekennzeichnet wurden, so quittiert der Compiler dies mit einer Warnung.
`@Deprecated` ähnelt dem aus früheren Java Versionen bekannte JavaDoc Tag `@deprecated`. Der Unterschied wird das an einem Code-Beispiele deutlich.

```
1  package java.util;
2  public class Date
3      implements java.io.Serializable, Cloneable, Comparable<Date>{
4  //...
5      /**
6       * [...]
7       * @deprecated As of JDK version 1.1, [...]
8       */
9      @Deprecated
10     public Date(int year, int month, int date) {
11         this(year, month, date, 0, 0, 0);
12     }
13 //...
14 }
```

Zwar hat der Java 1.4 Compiler bereits Warnungen produziert, wenn Methoden, Klassen oder Pakete genutzt wurden, die im JavaDoc als `@deprecated` gekennzeichnet waren. Jedoch sind Kommentare eigentlich durch den Compiler zu ignorieren. Daher ist der neue Weg „sauberer“, da hier die Methode selbst als *deprecated* gekennzeichnet wird, so dass sich der Compiler nicht mehr den Kommentar verarbeiten muss. Die Annotation `@Deprecated` wird auch mit in die JavaDoc Dokumentation aufgenommen allerdings nur als Annotation. Wie es dazu kommt wird im Kapitel *Meta-Annotationen* erläutert. Das JavaDoc-Tag `@deprecated` wird weiterhin verwendet jedoch nur zu Dokumentationszwecken.

@Override

`@Override` ist eine Annotation, welche ausschließlich auf Methoden anwendbar ist. Sie wird verwendet um Methoden zu kennzeichnen, die eine andere überschreiben soll. Tut eine gekennzeichnete Methode dies nicht, so gibt der Compiler eine Warnung aus. Das soll verhindern, dass Programmierer versehentlich eine Methode überladen anstatt sie zu überschreiben. Durch diese Annotation können solche Fehler früh erkannt werden. Zum Beispiel könnte ein Programmierer folgende `equals()` Methode schreiben.

```
1 public boolean equals(Foo that) {
2   //...
3 }
```

Gemeint war aber eigentlich diese:

```
1 public boolean equals(Object that) {
2   //...
3 }
```

Ohne die `@Override` Annotation würden beide Methoden ohne Probleme compiliert werden. Nutzt man aber `@Override`, so wird schon vom Compiler eine Warnung generiert. So würde folgendes Beispiel

```
1  public class Test {
2  /**
3    * Überschreibt die <code>equals</code> Methode
4    * von <code>Object</code>
5    */
6    @Override
7    public boolean equals( Test t) {
8      return true;
9    }
10 }
```

diese Meldung verursachen:

```
Test.java:3: method does not override a method from its superclass
        @Override
         ^
1 error
```

Wird jedoch eine Methode mit `@Override` versehen, die eine Methode eines Super-Intefaces überschreibt, so generiert der Compiler eine Fehlermeldung. Der Grund hierfür ist, dass die Methoden eines zu implementierenden Interfaces überschrieben werden müssen, daher ist eine zusätzliche Verwendung von `@Override` überflüssig. Daraus resultiert auch, dass `@Override` nicht innerhalb von `interface`-Deklarationen verwendet werden kann.

@SupressWarnings

Mithilfe der `@SupressWarnings` Annotation können, wie der Name bereits sagt, Warnungen des Compilers unterdrücken werden. Diese Annotation hat als Parameter einen Array von Strings.
`@SuppressWarnings(value = {S1, ... , Sk})`.
Die Strings `S1` bis `Sk` sind Bezeichner der zu unterdrückenden Warnungen. Wie diese Warnungen im einzelnen heißen, sollte vom Hersteller des Compilers spezifiziert werden. `@SupressWarnings` kann auf Typen, Felder, Methoden, Parameter, Konstruktoren sowie lokale Variablen angewendet werden.
Als Beispiel wäre denkbar, dass ein Programmierer folgende Zeilen geschrieben hat.

```
1 import java.util.Date
2 public class Test {
3   public void someMethod() {
4     set.add(new Date(104,8,11));
5   }
6 }
```

Worauf der Compiler folgende Warnung erzeugt:

```
Test.java:20: warning: [deprecation] Date(int,int,int) in
java.util.Date has been deprecated
                set.add(new Date(104,8,11));
                        ^
1 warnings
```

Ist der Programmierer sicher, dass er die Methode nutzen will, obwohl sie als *deprecated* gekennzeichnet ist, kann er die Warnung auf folgende Weise unterdrücken:

```
1 import java.util.Date
2 public class Test {
3         @SuppressWarnings({"deprecation"})
4   public void someMethod() {
5     set.add(new Date(104,8,11));
6   }
7 }
```

4 Eigene Annotationen

Natürlich lassen sich auch eigene Annotationen schreiben, will man z.B. Methoden oder Klassen als „unfertig“ kennzeichnen, so kann man das in einem JavaDoc Kommentar tun. Allerdings wäre es schwierig alle diese unfertigen Programmteile wieder zu finden, da Kommentare keinem Standard entsprechen. An dieser Stelle bieten Annotationen einen Ausweg. So kann man eine Annotation mit dem Namen `@InProgress` definieren und mit dieser dann alle Programmteile versehen, die noch eine Veränderung benötigen. Das könnte dann wie folgt aussehen:

```
1 /**
2  * Dies ist eine Marker-Annotation welche anzeigt, dass eine
3  * Methode oder Klasse noch nicht fertig ist
4  */
5 @Target({ElementType.TYPE,ElementType.METHOD,ElementType.CONSTRUCTOR})
6 public @interface InProgress {}
```

Zeile 5 bewirkt, dass diese Annotation nur auf Typen, Methoden und Konstruktoren anwendbar ist. Mehr dazu im Kapitel *Meta-Annotationen*
Mit dieser Annotation könnte dann z.B. eine Methode markiert werden.

```
1 public class Test
2 @InProgress
3 public Object makeFancyTings(Object o) {
4         return o;
5 }
```

Hierfür könnte man ein Tool schreiben, welches den Quellcode eines Projektes parst und die Entwickler darüber informiert, welche Teile des Projektes noch zu bearbeiten sind.

Dieses Beispiel lässt sich natürlich beliebig erweitern. Man könnte sich eine `@TODO` Annotation vorstellen, welche als Parameter einen Text erhält in dem steht, was genau zu tun ist. Und auch diese Annotation könnte von einem externen Tool ausgewertet werden.

```
/**
* Annotation um anzuzeigen, dass und was noch
* getan werden muss.
*/
public @interface TODO {
  String value() default "[nothing]";
}
```

```
public class Test {
  @TODO("Hier muß noch eine Menge getan werden")
  public Object makeFancyTings( Object o) {
    return o;
  }
}
```

Erweitert man das Beispiel, so bekommt man eine Annotation, die vielseitiger einsetzbar ist:

```
public @interface GroupTODO {
  enum Severity {CRITICAL, IMPORTANT, TRIVIAL};

  Severity severity() default Severity.IMPORTANT;
  String item();
  String assignedTo();
  String dateAssigned();
}
```

Mit dieser Annotation lassen sich Programmteile markieren, an denen noch zu arbeiten ist oder in denen ein Fehler aufgetreten ist. Es ließe sich ein Schärfegrad festlegen und diese Aufgaben könnten verschiedenen Personen zugewiesen werden. Ein externes Tool kann diese Annotation auswerten und z.B. die Programmierer, denen ein Problem zugewiesen wurde, per Email benachrichtigt, wenn sie es nicht binnen eines gewissen Zeitraums behoben wurde.

5 Meta-Annotationen

Meta-Annotationen sind nichts anderes als Annotationen, welche auf Annotations-Typen angewendet werden. Im J2SDK 1.5 Paket `java.lang.annotation` werden folgende mitgeliefert:

Annotation	Anwendbar auf	Wirkung
Inherited	Annotationen	Spezifiziert, dass diese Annotation, wenn sie auf eine Klasse angewendet wird, automatisch an deren Subklassen vererbt wird.
Documented	Annotationen	Spezifiziert, dass diese Annotation in die Dokumentation von annotierten Elementen eingebunden werden soll.
Target	Annotationen	Spezifiziert die Elemente, auf die diese Annotation angewendet werden kann.
Retention	Annotationen	Spezifiziert, wie lange diese Annotation beibehalten wird.

vergl. [HC05] Seite 1153

@Inherited

Annotiert man einen Annotations-Typen mit `@Inherited`, so wird durch den Compiler sichergestellt, dass Klassen, die mit dieser Annotation annotiert wurden, diese an ihre Kindsklassen vererben. Es macht also Sinn , die `@InProgress` Annotation mit `@Inherited` zu annotieren. `@Inherited` hat nur eine Wirkung auf Annotationen, die sich auf Klassen beziehen.

```
1 @Inherited
2 @Target({ElementType.TYPE,ElementType.METHOD,ElementType.CONSTRUCTOR})
3 public @interface InProgress {}
```

```
1 @InProgress
2 public class Animal {
3         private String name;
4         public String getName() {  return name;  }
5         public void   setName( String n) {  name = n; }
6 }
```

```
1 public class Dog extends Animal {
2 }
```

Die Klasse `Dog` besitzt somit auch die Annotation `@InProgress`.
Auf diese Art kann man sich Annotationen schaffen, die analog zu Markerinterfaces arbeiten.

@Documented

`@Documented` ist eine Annotation, welche von Dokumentationswerkzeugen, wie z.B. JavaDoc, ausgewertet wird. Solche Dokumentationswerkzeug sorgen dafür, dass diese Annotation, so wie die anderer *Modifizierer* (z.B. `protected`, `static`) in die Dokumentation mit eingehen. Nehmen wir als Beispiel die `@GroupTODO` Annotation. Wird diese mit `@Documented` versehen.

```
1  @Documented
2  public @interface GroupTODO {
3    public enum Severity {CRITICAL, IMPORTANT, TRIVIAL};
4
5    Severity severity() default Severity.IMPORTANT;
6    String item();
7    String assignedTo();
8    String dateAssigned();
9  }
```

So stehen die Informationen, die dem annotierten Element über `@GroupTODO` gegeben wurden, in der Dokumentation. Annotationen, die durch die Verwendung von `Inherited` geerbt wurden, stehen nicht in der Dokumentation.

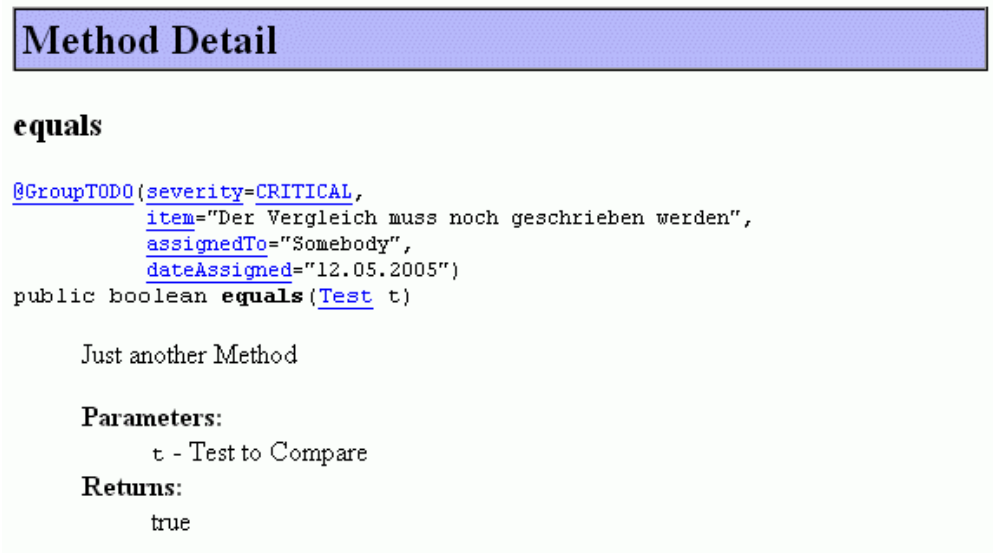

Abbildung 1: JavaDoc

@Target

Wendet man `@Target` auf eine Annotation an, so schränkt man damit die Arten der Elemente ein, auf die diese Annotation angewendet werden kann. `@Target` besitzt einen Parameter, welcher ein Array von `java.lang.annotation.ElementType` ist.
`@Target(value = {T1, ... , Tk})`
`java.lang.annotation.ElementType` ist ein Aufzählungstyp, welcher alle möglichen Elemente beinhaltet auf die Annotationen angewendet werden können.

ElementType	**Anwendbar auf**
ANNOTATION_TYPE	Typendeklarationen von Annotationen
PACKAGE	Pakete
TYPE	Klassen (einschließlich `enum` und Interfaces (einschließlich Annotations-Typen)
METHOD	Methoden
CONSTRUCTOR	Konstruktoren
FIELD	Felder (einschließlich `enum`-Konstanten)
PARAMETER	Parameter von Methoden oder Konstruktoren
LOCAL_VARIABLE	lokale Variablen

vergl. [HC05] Seite 1154

Wird die Art der möglichen „Ziele“ einer Annotation durch `@Target` eingeschränkt, kontrolliert dies der Compiler. Wendet man eine Annotation auf unzulässige Elemente an, führt das zu einem Compiler-Fehler. Eine Annotation ohne `@Target`-Einschränkung lässt sich auf alle Elemente anwenden.

@Retention

Die Annotation `@Retention` besitzt einen Parameter vom Typen `java.lang.annotation.RetentionPolicy` und veranlasst den Compiler, je nach dem was als Parameter übergeben wurde, diese Annotation nach dem compilieren zu verwerfen, sie in die `.class`-Datei zu laden und sie dort u.U. mit einem Flag zu versehen, so dass diese Annotation zur Laufzeit über das Reflection-API auswertbar ist. Bei dem Parameter handelt es sich um einen Aufzählungstypen. Die folgende Tabelle zeigt die möglichen Ausprägungen des Typen und deren Bedeutung für die `@Retention` Annotation.

ElementType	**Anwendbar auf**
SOURCE	Annotationen werden nicht in die Klassendateien eingebunden.
CLASS	Annotationen werden in die Klassendatei eingebunden, aber nicht von der VM geladen.
RUNTIME	Annotationen stehen zur Laufzeit über das Reflection-API zur Verfügung.

vergl. [HC05] Seite 1154

6 Ausblick

Wie Eingangs beschrieben können Annotatioen verwendet werden um zusätzliche Dateien oder gar neuem Quellcode zu generieren. Sollen die Annotationen auf Quellcode-Ebene verarbeitet werden bietet sich hierfür das Kommandozeilen-Tool apt an. Diesem Tool wird eine Liste von Quelldateien übergeben, welche nach Annotationen durchsucht werden. Die gefundenen Annotationen werden mit Hilfe von Annotations-Prozessoren verarbeitet. Wurden neue Quelldateien erzeugt, wiederholt sich der Vorgang.
Mit apt und entsprechenden Annotations-Prozessoren kann der Umgang mit der „Enterprise Edition“ von Java, welche bekannt dafür ist, dass Programmierer eine Menge Standardcode schreiben müssen, erleichtert werden. Der Standard-Code lässt sich durch die Verwendung von Annotationen automatisch generieren. Aber nicht nur neuer Code, sondern auch XML-Deskriptoren, Eigenschaftsdateien, HTML-Dokumentationen usw. lassen sich generieren.

Das Reflection-API bietet die Möglichkeit zur Laufzeit auf Elemente eines Programms zuzugreifen und ihre Ausprägung zu analysieren. Dies gilt auch für Annotationen soweit sie zur Laufzeit noch zur Verfügung stehen. So lassen sich z.B. Objekte unter zu Hilfenahme eines Object-Relational-Mappers in einer Datenbank speichern, wenn Ihre Eigenschaften in entsprechenden Annotationen beschrieben sind.

Annotationen sind ein leistungsfähiges Instrument, allerdings bergen sie auch Gefahren. Werden Annotationen zu freizügig eingesetzt, kann es zu Code kommen, dessen Inhalt nur noch schwer verständlich ist. Ähnliche Erfahrungen kann man machen, wenn man den C-Präprozessor allzu freizügig verwendet. Daher sollte man bei der Verwendung von Annotationen darauf achten, dass der Quellcode sich auch ohne Annotationen fehlerfrei compilieren lässt. Dadurch beugt man sogenannte „Meta-Quellen“ vor, die nur noch durch die Verwendung eines Annotations-Prozessors in gültigen Code überführt werden können.

Literatur

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Sun Developer Network (SDN), 2005. http://java.sun.com/docs/books/jls/.

[HC05] Cay Horstmann and Gary Cornell. *Core Java 2 Bd.2 Expertenwissen*. Addison-Wesley, München, 2005.

[Now04] Johannes Nowak. *Fortgeschrittene Programmierung mit Java 5*. Dpunkt Verlag, 2004.