Frege Teil 2: Ein Haskell für die JVM ▸ 18

JAVA Mag

Deutschland €9,80 Österreich €10,80 Schweiz sFr 19,50 Luxemburg €11,15 12.2015

Javamagazin

Java • Architekturen • Web • Agile

www.javamagazin.de

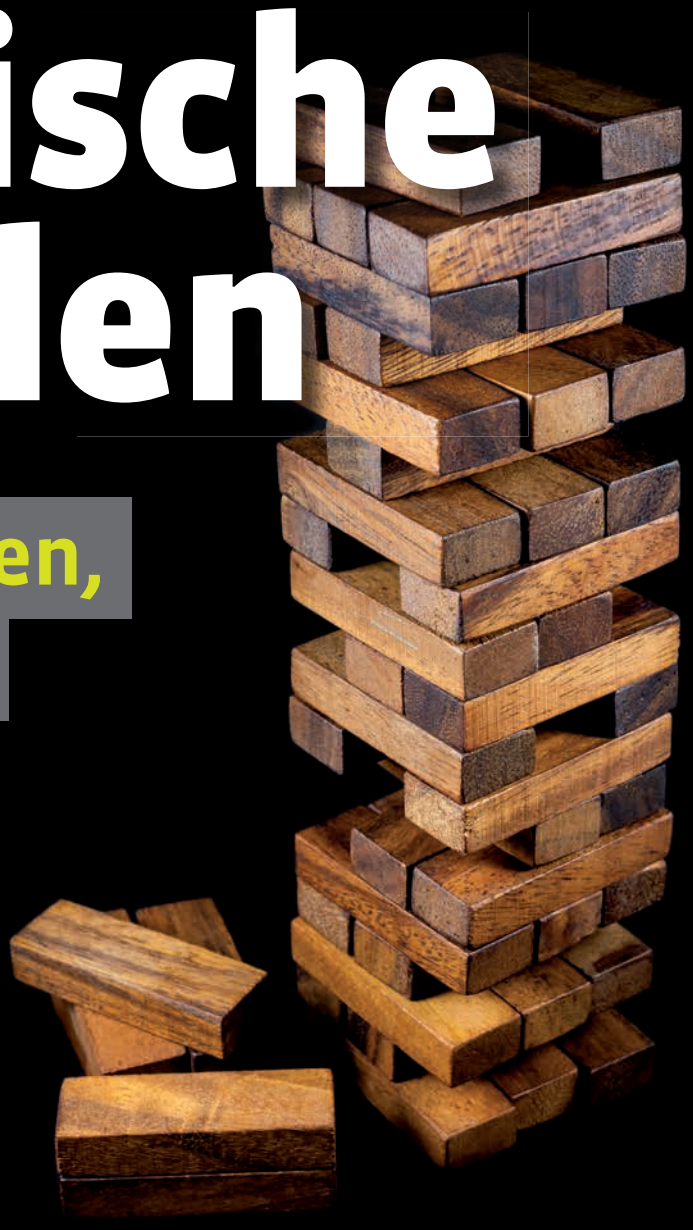
Effective Java
Der Vergleich: reduce() und collect() ▸11

Hystrix in Action
Der Weg zu robuster Software ▸44

MQTT
Sicherheit im IoT durch offene Standards ▸102

Technische Schulden

Baumängel finden, bevor es kracht



4 194586 709801 12

Die Last der Lasttests ▸73

Der fehlende Android-Guide: Betreiben einer App ▸107

Prüfbarkeit von Websystemen ▸84

Wie Taglet- und Doclet-APIs die Softwaredokumentation vereinfachen können

Von Taglets und Doclets

Wenn das Standard-Javadoc nicht genügt oder gefällt, wenn es um die Dokumentation von Java-Klassen geht, wird oft auf das dem JDK beiliegende Javadoc-Werkzeug zurückgegriffen. Auch wenn bereits mit dessen Standardeigenschaften sehr viel umgesetzt wird, gibt es Situationen, in denen die Fähigkeiten dieses Tools nicht ausreichen oder die Art der HTML-Reports nicht gefällt. Nicht selten beginnt in solchen Fällen die Suche nach alternativen Dokumentationswerkzeugen, die vorzugsweise mit Javadoc-Inhalten umgehen können. Dass es für Javadoc Erweiterungen von Drittanbietern gibt, ist weniger bekannt. Dass mit Taglets und Doclets zudem APIs existieren, mit denen man seine eigenen Anpassungen vornehmen kann, ist vielen ebenfalls unbekannt. Wir wollen ein wenig Licht ins Dunkel bringen, indem wir uns anhand eigener exemplarischer Erweiterungen den APIs nähern.

von Sven Hofrichter

Für jene, die bisher noch nicht oder nur rudimentär mit Javadoc gearbeitet haben, sei auf die von Oracle bereitgestellten Tutorials und FAQs verwiesen, die einen schnellen Einblick in die kleine Welt des Javadocs bieten. Die verschiedenen Seiten liegen teilweise in leicht verteilter Form und vor allem lose gekoppelt vor, was durchaus einem zeitgemäßen Programmierstil folgt, dem Autor dieses Artikels allerdings die Zusatzaufgabe auferlegt, sicherheitshalber alle Javadoc-Standardtags tabellarisch zusammenzufassen (Tabelle 1), um die Einarbeitung oder die Auffrischung des Wissens etwas zu vereinfachen. Sie ersetzt die offizielle Dokumentation jedoch nicht (siehe [1] und insbesondere [2]).

Die Verwendung der in der Tabelle gelisteten Tags zeigt das Beispiel aus Listing 1.

Da unser Artikel nicht auf die reine Verwendung von Javadoc abzielt, sondern vielmehr das Ausreizen seiner Funktionalität durch eigene Erweiterung zum Thema hat, wollen wir es bei dieser kurzen Einführung belassen und uns nun in spannendere Gefilde vorwagen. Dem werden wir gerecht, indem wir uns das API der *Taglets* und der *Doclets* genauer anschauen und mit Beispielen untermauern. Das vollständige Beispiel wird über ein Repository [7] bereitgestellt und kann als Ausgangsbasis für eigene Erweiterungen verwendet werden oder aber einfach nur als Nachschlagewerk dienen.

Taglets

Taglets werden durch so genannte Javadoc-Tags (in Kommentaren platzierte Schlagworte mit vorangestelltem @-Symbol) repräsentiert, zu denen vor allem die in Tabelle 1 gezeigten Vertreter zählen, deren Verwendung in Listing 1 kurz angedeutet wurde. Jedes dieser Tags hat seine eigene Bedeutung und wurde mit einem Standardverhalten verbunden, sodass sie vom Javadoc-Interpreter entsprechend ihrer Implementierung umgesetzt werden. Dieses Verhalten lässt sich mit sehr geringem Aufwand an eigene Bedürfnisse anpassen, bzw. es lassen sich auf sehr einfache Weise auch neue Tags definieren. Das kann in zweierlei Form erfolgen:

1. Definition eines neuen Taglets durch Parametrisierung des Aufrufs von Javadoc
2. Implementierung des Interface *com.sun.tools.doclets.Taglet*

Die einfachste und schnellste Form der Änderung bestehender oder die Bereitstellung neuer Taglets erfolgt durch Parametrisierung des *javadoc*-Aufrufs. Als Parameter werden dabei pro Tag drei Informationen angegeben, die jeweils durch einen Doppelpunkt voneinander getrennt sind:

```
javadoc -tag <Tagname>:<Platzierung>:<Überschrift>
```

Die im Beispiel verwendeten Platzhalter haben folgende Bedeutung:

1. <*Tagname*> ist der Name des Tags im Kommentar der Java-Sourcen
2. <*Platzierung*> definiert, in welchen Bereichen das Tag zulässig ist oder ob es überhaupt ausgegeben

werden soll (Mehrfachnennung ist ohne Trennzeichen möglich):
- X (Tag deaktivieren)
- a (alle nachfolgenden Eigenschaften aktivieren)
- o (Platzierung in der Übersicht/Overview)
- p (Verwendung in *packages*)
- t (Verwendung bei „Types“, also Klassen, Interfaces)
- c (Verwendung bei Konstruktoren)
- m (Tag ist bei der Dokumentation von Methoden zulässig)
- f (Tag darf bei Feldern/Field verwendet werden)

3. <*Überschrift*> ist die Textpassage, dem der jeweilige Kommentar vorangestellt wird.

Wie man sieht, ist diese Form der Erweiterung äußerst trivial, jedoch lassen sich auf diese Weise nur Blocktags definieren, die nicht innerhalb von Dokumentationstexten platzierbar sind.

Diese Restriktion wollen wir so natürlich nicht akzeptieren und greifen deshalb auf ein etwas aufwändigeres Mittel zurück, indem wir das Interface *com.sun.tools.doclets.Taglet* implementieren, dem einzigen Vertreter des Taglet-API. Das genannte Interface befindet sich im *[JAVA_HOME]/lib/tools.jar*, einer Datei, die nicht im Standard-Classpath des *javac*-Compilers enthalten ist. Aus diesem Grund muss diese Bibliothek beim Kompilieren explizit mittels *-classpath* bekanntgegeben werden: *javac -classpath <path_to>/tools.jar <Taglet-Javadateien>*.

Durch die Implementierung eines Taglets lassen sich Eigenschaften festlegen, die mit der eben gezeigten, rein kommandozeilenbasierten Variante nicht möglich sind.

Javadoc-Tag	Seit Java-Version	#	o	p	i	c	m	f	con	inl	Kurzbeschreibung
@author	1.0	1	x	x	x	x					Name des Autors
{@code}	1.5									x	Formatiert die Textpassage als Code
{@docRoot}	1.3		x	x	x	x	x	x	x	x	Relativer Pfad zum Wurzelverzeichnis im generierten HTML-Ergebnis von Javadoc
@deprecated	1.0	9			x	x	x	x	x		Markiert eine Methode als veraltet, die in kommenden Releases ausgebaut wird
@exception	1.0	5					x		x		siehe *@throws*
{@inheritDoc}	1.4						x		x	x	„kopiert“ bei abgeleiteten Klassen/überschriebenen Methoden die Javadoc-Definition (die Beschreibung selbst zzgl. der Tags *@return*, *@param* and *@throws*)
{@link}	1.2		x	x	x	x	x	x	x	x	Fügt einen Verweis auf eine Klasse, Methode oder andere Eigenschaft ein
{@linkplain}	1.4		x	x	x	x	x	x	x	x	Analog zu *@link*, jedoch mit der Möglichkeit, ein Label (Linktext) zu definieren
{@literal}	1.5									x	Fügt Text hinzu, der nicht als HTML-Markup interpretiert wird
@param	1.0	3					x		x		Dient der Beschreibung eines Methodenparameters
@return	1.0	4					x				Dient der Beschreibung eines Methodenergebnisses bei Nicht-void-Methoden
@see	1.0	6	x	x	x	x	x	x	x		Fügt einen Verweis auf eine Klasse/Methode ein, allerdings in einem separaten Abschnitt (nicht inline)
@serial	1.2	8		x	x	x		x			Markiert serialisierbare Eigenschaften/Objekte*
@serialData	1.2						x		x		Markiert serialisierbare Datenstruktur*
@serialField	1.2							x			Markiert serialisierbare Eigenschaften vom Typ *ObjectStreamField**
@since	1.1	7	x	x	x	x	x	x	x		Definiert, seit wann die Eigenschaft/Methode/Klasse verfügbar ist und bezieht sich auf die API-/Programmversion
@throws	1.2	5					x		x		Beschreibt die Exception, die von der Methode im Fehlerfall geworfen wird
{@value}	1.4							x		x	Platzhalter für den Wert einer Konstante (somit nur bei statisch-markierten Feldern anwendbar)
@version	1.0	2	x	x	x	x					Definiert die aktuelle Version bzw. eine Versionshistorie

Bedeutung der Spalten: p=package, c=class, i=interface, m=method, f=field, con=constructor , inl=Inlinetag, # empfohlene Reihenfolge bei Verwendung der Tags im Kommentar
Quellen: [3], [4]
*) siehe auch [5], [6]

Tabelle 1: Javadoc-Standardtags

Listing 2 zeigt, wie man beispielsweise den *@author*-Tag überschreibt. Analog zu diesem Beispiel kann man auch neue Tags erstellen. Entscheidend ist dabei nur der Taglet-Name, der sich nicht aus dem Klassennamen ableitet, sondern durch den Rückgabewert der Methode *getName()* bestimmt wird. Darüber hinaus wird über die Methode *isInlineTag()* unterschieden, ob es sich um ein Inline-Tag oder ein Blocktag handelt. Diese Eigenschaft definiert zudem, ob im Fall von Inline-Tags *toString(Tag)* oder bei Blocktags *toString(Tag[])* vom Interpreter angesprochen wird.

So weit, so gut. Aber wie erfährt Javadoc von unseren Taglets? Das Einbringen der kompilierten Taglets in den Classpath genügt nicht, da Javadoc seine Umdefinitionen

Listing 1: Beispiel einer Javadoc-konformen Klasse

```
/**
 * Bedeutung der Klasse. Der erste Satz sollte mit
 * einem Punkt beendet werden, da der Text in der
 * Summary wiederverwendet wird.
 *
 * @author Sven Hofrichter
 * @version 1.1
 * @version 1.2
 * @version 1.3
 * @since 1.1
 */
public class JavadocBeispiel implements Serializable
{

 /**
  * Definiert den Wert {@value} als Konstante.
  */
 public final static int KONSTANTE = 1;

 /**
  * Default Constructor der Klasse, der intern die
  * Konstante {@value #KONSTANTE} verwendet, um ...
  */
 public JavadocBeispiel() {...}

 /**
  * Diese Methode verarbeitet die übergebene
  * Collection ...
  *
  * @param param1 ist ...
  * @param param2 definiert die ...
  */
 protected void beispielMethode1(String param1, int
                                          param2) {...}

 /**
  * Diese Methode verarbeitet die übergebene
  * Collection ...
  *
  * @param param1 definiert eine Collection, die ...
  * @return das Ergebnis, welches ...
  * @throws IOException im Falle eines Fehlers
  * bei ...
  */
 protected Map<String, String> beispielMethode2(
 Collection<Integer> param1) throws IOException {...}

 /**
  * Die Methode wurde überschrieben, weil ...
  * {@inheritDoc}
  */
 @Override
 public String toString() {...}
}
```

Listing 2

```
public class AuthorTaglet implements Taglet {
  protected static final String TAGLET_NAME =
                                              "author";
  public String getName() {
    return TAGLET_NAME;
  }
  public static void register(Map tagletMap) {
    tagletMap.put(TAGLET_NAME, new
                                    AuthorTaglet());
  }
  public boolean inConstructor() {
    return false;
  }
  public boolean inField() {
    return false;
  }
  public boolean inMethod() {
    return false;
  }
  public boolean inOverview() {
    return true;
  }
  public boolean inPackage() {
    return true;
  }
  public boolean inType() {
    return true;
  }
  public boolean isInlineTag() {
    return false;
  }
  public String toString(Tag tag) {
    return renderAuthor(tag);
  }
  public String toString(Tag[] tags) {
    StringBuffer sb = new StringBuffer();
    if (tags != null && tags.length > 0) {
      sb.append("<dt style=\"font-weight:bold;
                          text-decoration:underline\">");
      sb.append(tags.length > 1 ? "Autoren" :
                                                "Autor");
      sb.append(":</dt>");
      for (int i = 0; i < tags.length; i++) {
        sb.append("<dd>");
        sb.append(renderAuthor(tags[i]));
        sb.append("</dd>");
      }
    }
    return sb.toString();
  }
  private String renderAuthor(Tag tag) {
    String author = toString(tag);
    if (author == null) {
      return "";
    }
    Doc doc = tag.holder();
    String className = doc.name();

    // E-Mail-Adressen werden in mailto-Links
    // umgewandelt:
    author = author
    .replaceAll(
      "(^|[^a-zA-Z0-9_.+-])([a-zA-Z0-9_.+-]+
              @[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]{2,4})
                                  ($|[^a-zA-Z0-9-.])",
      "$1<a href=\"mailto:$2?subject=Anmerkung zur
                                          Javaklasse '"
        + className + "'\">$2</a>$3");
    return author;
  }
}
```

nicht aus dem Classpath bezieht, sondern hierfür eigene Parameter bereitstellt. Listing 3 zeigt die minimal erforderliche Parametrisierung, um das Standardverhalten von Javadoc um die eigenen Erweiterungen aufzuwerten.

Die bisher gezeigten Javadoc-Aufrufe (mit oder ohne eigene Taglet-Implementierung) lassen sich auch, wie in Listing 4 gezeigt, kombinieren.

Wem das ständige Getippe von langen Javadoc-Aufrufen zu viel ist, der kann etwas Zeit sparen, indem er sich ein Shell-Skript für die jeweilige Plattform schreibt oder alle Taglets in eine Konfigurationsdatei, wie in Listing 5 (siehe auch [8]) zu sehen, auslagert und diesem dem Javadoc-Aufruf (Listing 6) mitgibt. Wichtig ist, dass jeder Kommandozeilenparameter in der Datei nicht hintereinander, sondern untereinander definiert wird, wobei die zum Parameter gehörenden Werte/Angaben in die Zeile des Parameters gehören.

Aus Listing 6 wird transparent, dass dem Pfad zur jeweiligen Konfigurationsdatei ein @-Symbol voranzustellen ist. Diese Art der Parametrisierung kann im Übrigen für alle Javadoc-Parameter angewendet werden und bezieht sich nicht ausschließlich auf die Definition von Taglets oder den gleich folgenden Doclets.

Mit einem Beispiel zur Integration der eigenen Javadoc-Erweiterungen in ein Build-Tool (hier Maven) soll das überschaubare Thema Taglets abgeschlossen werden. Listing 7 zeigt die exemplarische Einbindung unserer kleinen Taglet-Sammlung in Maven. Weiterführende Informationen und vor allem mehr Beispiele zur Einbindung in Maven können unter [9] bezogen werden.

Listing 3

```
$> javadoc -tagletpath <path_to_our_taglets>/classes -taglet net.hofrichter.javamag.
    javadoc.taglets.NotizTaglet -taglet net.hofrichter.javamag.javadoc.taglets.AuthorTaglet
                                                                    <path_to>/*.java
```

Listing 4

```
$> javadoc -tagletpath <path_to_our_taglets>/classes -taglet net.hofrichter.javamag.
                javadoc.taglets.NotizTaglet -taglet net.hofrichter.javamag.javadoc.taglets.
                        AuthorTaglet -tag example:a:"Beispiel: " <path_to>/*.java
```

Listing 5

```
-tagletpath <path_to>/classes
-taglet net.hofrichter.javamag.javadoc.taglets.NotizTaglet
-taglet net.hofrichter.javamag.javadoc.taglets.AuthorTaglet
-tag example:a:"Beispiel: "
```

Listing 6

```
javadoc @beispiel-config-1 @<path_to>/beispiel-confg-2 ...
```

Doclets

Wer Javadoc parameterlos oder zumindest ohne Doclet-Parameter (also ohne Parameter mit Präfix *-doclet*) startet, dem werden Standard-HTML-Reports generiert, mit denen unter anderem auch das Javadoc der Programmiersprache bereitgestellt wird. Dabei handelt es sich um das Ergebnis des im Javadoc-Werkzeug integrierten *com.sun.tools.doclets.standard.Standard*-Doclets nebst zugehöriger Klassen. Apropos *Standard-Doclet*: Die in der Javadoc-Dokumentation aufgeführten Parameter werden teilweise von der Doclet-Implementierung bereitgestellt, sodass diese bei einer eigenen Umsetzung nicht automatisch unterstützt werden und vom Doclet-Entwickler bei Bedarf nachgebaut werden müssen. Es besteht natürlich die Möglichkeit, von den Klassen des *Standard*-Doclets zu erben, was aufgrund der vielen statischen Methoden allerdings ein etwas eigenartig anmutendes Codelayout zur Folge hat.

Auch ist der Ablauf analog dem Taglet-API zu verstehen. Javadoc parst alle Java-Sourcen und erstellt daraus ein Metamodell, das mit dem Doclet-API in das gewünschte Ausgabeformat umgewandelt wird. Dabei

Listing 7

```
<project>
  ...
  <reporting> <!-- oder <build> -->
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
        <version>2.10.3</version>
        <configuration>
          ...
          <taglets>
            <taglet>
              <tagletClass>package.to.AuthorTaglet</tagletClass>
            </taglet>

            <taglet>
              <tagletClass>package.to.NotizTaglet</tagletClass>
            </taglet>
            ...
          </taglets>
          <tagletArtifact>
            <groupId>group-id-Taglet</groupId>
            <artifactId>Custom-Taglet-artifact-id</artifactId>
            <version>Custom-Taglet-version</version>
          </tagletArtifact>
          ...
        </configuration>
      </plugin>
    </plugins>
    ...
  </reporting> <!-- oder </build> -->
  ...
</project>
```

reicht das Spektrum des API von kleinen Erweiterungen/Änderungen bis hin zur Implementierung eines vollständig neuen Ausgabeformats (bspw. PDF). Für unser Beispiel wurde ein Ansatz gewählt, der bewusst nicht von der Standardimplementierung erbt, aber ebenfalls HTML-Seiten ausgibt. Das unter [7] veröffentlichte Beispiel ist hoffentlich auch in Bezug auf Doclets klar und verständlich gestaltet.

Der korrekte Einstiegspunkt

Eine kurze, aber dennoch gute Einführung in das Thema Doclets stellt Oracle unter [10] selbst bereit. Auch wenn mancher weiterführende Link ins Leere führt oder nur die gleiche Seite anspricht, dient diese Anlaufstelle als Ausgangsbasis für diesen Abschnitt des vorliegenden Artikels.

Wie bereits angedeutet, ist die Signatur bestimmter Klassen dank ihrer statischen Methoden etwas gewöhnungsbedürftig. Wer also nur kleine Erweiterungen einbringen möchte, muss sich aus diesem Grund intensiver mit den Klassen der Basisimplementierungen beschäftigen und wird manchmal gezwungen, unnötig andere Teile zu überschreiben, nur um an einem bestimmten Punkt mit der eigenen Implementierung beginnen zu können.

Dieser unnötigen Komplexität gehen wir in unserem Beispiel aus dem Weg und verzichten bei der in Listing 8 gezeigten *Main*-Klasse auf das Erben vom Standard-Doclets.

Wenn wir keine Kommandozeilenparameter unterstützen wollen, genügt die Bereitstellung der Methode *public static boolean start(com.sun.javadoc.RootDoc root)*. Damit haben wir den Ausgangspunkt unserer Eigenkreation eigentlich schon vollständig definiert. Alles Weitere wird durch Verarbeitung der RootDoc-Informationen realisiert. Wem diese Information genügt, kann das Weiterlesen an diesem Punkt einstellen. Allerdings fehlt das eine oder andere Detail, denn wie bereits erwähnt, sind wir beispielsweise noch immer ohne Unterstützung von Parametern unterwegs. Auch haben wir noch keinen ordentlichen Ausgabekanal für Informationen, Warnungen oder Fehler eingebunden, der vom Javadoc-Tool bereitgestellt wird und somit nicht *System.err* und *System.out* lautet.

Die Unterstützung von Parametern, die wir zudem vorvaldieren möchten, wird mit den Methoden *public static int optionLength(String option)* und *public static boolean validOptions(String options[][], com.sun.javadoc.DocErrorReporter reporter)* realisiert, wobei erstgenannte die Anzahl möglicher Werte zurückgibt (min. 1, da der jeweilige Parameter mitgezählt wird) und die zweite die Prüfung der Parameterwerte übernimmt. In der Signatur der zweiten Methode sehen wir zudem einen Parameter vom Typ *DocErrorReporter*. Diese Instanz ist unser Kommunikationskanal, wenn es darum geht, dem Anwender Informationen, Warnungen oder gar Fehlermeldungen zu vermitteln. Die *RootDoc*-Instanz der *start*-Methode implementiert im Übrigen ebenfalls dieses Interface, sodass auch hier die Methoden zur Kommunikation bereitstehen, die auf die Namen *printNotice*, *printWarning* und *printError* hören. Die Verwendung der letztgenannten *printError*-Methode inkludiert einen Abbruch des Javadoc-Prozesses. Listing 8 zeigt die auf die wesentlichen Details reduzierte Startklasse unseres Beispielprojekts, in dem wir zwar nicht, wie bereits erwähnt, vom Standard-Doclet erben, dafür aber von einer abstrakten Klasse namens *com.sun.javadoc.Doclet*. Das ist zwar nicht zwingend erforderlich, bietet aber die Möglichkeit, nach einem Update

Listing 8

```
public class BaseDoclet extends Doclet {
  public static boolean start(RootDoc root) {
    try {
      ClassDoc[] classDocs = root.classes();
      ...
      return true;
    } catch (Exception e) {
      root.printError("Failed to create javadoc! " + e.getLocalizedMessage());
    }
    return false;
  }
  public static boolean validOptions(String options[][], DocErrorReporter reporter) {
    boolean result = true;
    try {
      for (String[] option : options) {
        switch (option[0]) {
          case "-d":
          outputDir = new File(option[1]);
          break;
          ...
          default:
          reporter.printError("Unknown commandline option: " + option[0]);
          return false;
        }
      }
      ...
    } catch (Exception e) {
      reporter.printError("Commandline error: " + e.getLocalizedMessage());
      result = false;
    }
    return result;
  }
  public static int optionLength(String option) {
    switch (option) {
      case "-d":
      return 2;
      ...
      default:
      return 0;
    }
  }
  public static LanguageVersion languageVersion() {
    return LanguageVersion.JAVA_1_5;
  }
}
```

von Javadoc Änderungen im API ausfindig zu machen. Mit Ausnahme der *start*-Methode implementiert diese Klasse alle relevanten Methoden mit einem sinnvollen Defaultverhalten.

Die bisher nicht erwähnte Methode *public static LanguageVersion languageVersion()* definiert, welche Version der Sprache von unserem eigenen Doclet unterstützt wird. Anders als in der Klasse *Doclet* festgelegt, wollen wir in unserem Beispiel *Generics* unterstützen, weshalb wir *JAVA_1_5* zurückgeben.

Listing 9

```
public class ClassDocDoclet extends Doclet {
  public static boolean start(RootDoc root) {
    for (ClassDoc doc : root.classes()) {
      System.out.println("Name: " + doc.name());
      System.out.println("Eigenschaften: " + doc.modifiers());
      System.out.println("Kommentar: " + doc.commentText());
      System.out.println("Vollqualifizierter Name: " + doc.qualifiedTypeName());
      System.out.println("Packagename: " + doc.containingPackage().name());
      for (MethodDoc mdoc : doc.methods()) { ... }
        ...
    }
    return true;
  }
}
```

Da das Doclet-API aus dreißig Klassen bzw. Interfaces besteht und wir nicht auf jedes Element im Detail eingehen wollen, soll für eine vollständige Auflistung und Detailbeschreibung ein Verweis auf die Doclet-API-Dokumentation [11] genügen. In unserem Artikel konzentrieren wir uns auf Feinheiten und Unwegsamkeiten, die bei der Verwendung des API aus Sicht des Autors beachtenswert sind.

Fühle die Macht der Doclets

Unser Ausgangspunkt ist die *RootDoc*-Instanz, die uns vom Javadoc-Tool über die Methode *start(RootDoc)* eingereicht wurde und diverse Informationen bereitstellt. Eine Methode lautet *String[][] options()* und kam bereits zum Einsatz, als uns Javadoc zum Validieren der Kommandozeilenparameter über die Methode *validOptions(...)* aufforderte. Neben vier weiteren, hier nicht näher betrachteten Methoden wollen wir uns der *ClassDoc[] classes()* bedienen, um das Ergebnis, ein Array aus Metainformationen zu allen vom Javadoc-Aufruf erfassten Quellcodedateien, zu erhalten.

Jede *ClassDoc*-Instanz aus dem Array repräsentiert die Javadoc-Informationen einer Java-Klasse, die der Auswertung von Kommentaren dienlich sein können. Dazu gehören neben den reinen Kommentaren auch Signaturinformationen, wie Methoden, Felder, Package etc. Aus dem Pool an 25 *ClassDoc*-Methoden sollen uns fürs Erste all jene genügen, die die Signatur der jeweili-

Listing 10

```
public class WrapperTaglet implements Taglet {
  [...]
  public WrapperTaglet(String tagletClassName)
                                    throws Exception {
    Class<?> tagletClass = (Class<?>) Class.
                              forName(tagletClassName);
    Object instance = tagletClass.newInstance();
    this.tagletName = invoke(instance, "getName",
                                       String.class);
    this.blockCommentLabel = tagletName.
          substring(0, 1).toUpperCase() + tagletName.
                                       substring(1);
    this.isInlineTag = invoke(instance, "isInlineTag",
                                     Boolean.class);
    this.inConstructor = invoke(instance,
                        "inConstructor", Boolean.class);
    this.inField = invoke(instance, "inField", Boolean.
                                              class);
    this.inMethod = invoke(instance, "inMethod",
                                     Boolean.class);
    this.inType = invoke(instance, "inType", Boolean.
                                              class);
    this.inPackage = invoke(instance, "inPackage",
                                     Boolean.class);
    this.inOverview = invoke(instance, "inOverview",
                                     Boolean.class);
  }
  private static <T> T invoke(Object instance, String
methodName, Class<T> returnType) throws Exception {
    Method method = instance.getClass().
                                  getMethod(methodName);
    return (T) method.invoke(instance);
  }
  public String getName() {
    return tagletName;
  }
  public boolean inConstructor() {
    return inConstructor;
  }
  public boolean inField() {
    return inField;
  }
  public boolean inMethod() {
    return inMethod;
  }
  public boolean inType() {
    return inType;
  }
  public boolean inPackage() {
    return inPackage;
  }
  public boolean inOverview() {
    return inOverview;
  }
  public boolean isInlineTag() {
    return isInlineTag;
  }
  public String toString(Tag tag) {
    switch (tag.name()) {
      case "code": return "<code>" + tag.text() +
                                              "</code>";
      default: return tag.text();
    }
  }
  public String toString(Tag[] tags) {
    StringBuffer sb = new StringBuffer();
    if (tags != null && tags.length > 0) {
      sb.append("<dt style=\"font-weight:bold;
                        text-decoration:underline\">");
      sb.append(blockCommentLabel);
      sb.append(":</dt>");
      for (Tag tag : tags) {
        sb.append("<dd>").append(tag).append(
                                              "</dd>");
      }
    }
    return sb.toString();
  }
}
```

gen Klasse beschreiben und somit über die in Listing 9 verwendeten hinausgehen. Hierzu gehören insbesondere *ConstructorDoc[] constructors()*, *MethodDoc[] methods()* und *FieldDoc[] fields()*. Dabei handelt es sich jeweils um *ProgramElementDoc*-Instanzen, einem Interface, das vor allem Informationen zu Basismerkmalen bereitstellt, wozu man unter anderem Methoden wie *String name()* (der einfache Name des Java-Elements), *SourcePosition position()* (Position des Elements, dessen Kommentar wir in der aktuellen *ProgramElementDoc*-Instanz haben) oder *String commentText()* (von Tags bereinigt) bzw. *String getRawCommentText()* (Originalkommentar) zählen darf.

Neben den Möglichkeiten, auf Informationen zum Objekt selbst oder auf dessen Kindelemente zuzugreifen, existieren auch jene, die den Weg nach „oben" eröffnen – also die Eltern zugänglich machen. Bei *ClassDoc* sind es unter anderem Methoden, wie *Type superclassType()* (Vererbung) und *TypeVariable[] typeParameters()* bzw. *ParamTag[] typeParamTags()* (Generic-Types).

Listing 11

```
public class BaseDoclet extends Doclet {
  [...]
  private final static String[] DEFAULT_TAGLETS = new String[] {
    "com.sun.tools.doclets.internal.toolkit.taglets.CodeTaglet",
    "com.sun.tools.doclets.internal.toolkit.taglets.DeprecatedTaglet",
    "com.sun.tools.doclets.internal.toolkit.taglets.DocRootTaglet",
    "com.sun.tools.doclets.internal.toolkit.taglets.ExpertTaglet",
    "com.sun.tools.doclets.internal.toolkit.taglets.InheritableTaglet",
    "com.sun.tools.doclets.internal.toolkit.taglets.InheritDocTaglet",
    "com.sun.tools.doclets.internal.toolkit.taglets.LegacyTaglet",
    "com.sun.tools.doclets.internal.toolkit.taglets.LiteralTaglet",
    "com.sun.tools.doclets.internal.toolkit.taglets.ParamTaglet",
    "com.sun.tools.doclets.internal.toolkit.taglets.PropertyGetterTaglet",
    "com.sun.tools.doclets.internal.toolkit.taglets.PropertySetterTaglet",
    "com.sun.tools.doclets.internal.toolkit.taglets.ReturnTaglet",
    "com.sun.tools.doclets.internal.toolkit.taglets.SeeTaglet",
    "com.sun.tools.doclets.internal.toolkit.taglets.SimpleTaglet",
    "com.sun.tools.doclets.internal.toolkit.taglets.ThrowsTaglet",
    "com.sun.tools.doclets.internal.toolkit.taglets.ValueTaglet" };
    [...]
  private static void initDefaultTaglets(DocErrorReporter reporter) {
    for (String tagletName : DEFAULT_TAGLETS) {
      try {
        WrapperTaglet taglet = new WrapperTaglet(tagletName);
        taglets.put(taglet.getName(), taglet);
      } catch (Exception e) {
        reporter.printWarning("Standard-Taglet '" + tagletName
          + "' ist nicht verfügbar, weil die Initialisierung fehlschlug! "
          + e);
      }
    }
  }
}
```

Interessant wird es jedoch, wenn Klassen oder Methoden von Javadoc aufgrund ihrer „Sichtbarkeit" (also Scopes, wie *private*, *protected* oder der *default*-Scope) ausgesteuert wurden. Solche Elemente sind nämlich nicht im Ergebnis unseres Ausgangspunkts *ClassDoc[] RootDoc.classes()* enthalten. In diesem Fall erhalten wir die *Doc*-Instanzen ausschließlich dann, wenn sie von anderen Objekten referenziert werden und beispielsweise als *extends*-Ausdruck in einer anderen Klasse auftauchen, was in der Beispielimplementierung unter [7] anhand der abstrakten Klasse *AbstractTemplate* (mit standardmäßig ausgesteuertem *default*-Scope) gezeigt wird. Alternativ bestehen zwei weitere Möglichkeiten, die diese Filterung beeinflussen bzw. umgehen:

1. Javadoc mit Parametern starten, die dieses Verhalten steuern (*-private*, *-package*, *-protecte*)
2. Über die Package-Struktur per *RootDoc.specifiedPackages()* zu den Klassen navigieren

Das Ergebnis der zweitgenannten Option ist ein Array aus *PackageDoc*-Instanzen, die wiederum eine Methode namens *allClasses(boolean)* bereitstellen, über die sich die Filterung mit dem Parameter *false* gezielt ausschalten lässt.

Da fehlt doch etwas

Deutlich kniffliger wird es, wenn es um die Unterstützung von Taglets geht. Hier stellt das Doclet-API dem Entwickler zwar in Form des Interface *com.sun.javadoc.Tag* ein Abstraktionsmittel bereit, allerdings nur auf der Ebene, wie wir es von den *Doc*-Klassen kennen. Das Interface stellt nur Informationen bereit. Eine Implementierung, die sich in Form eines Taglets widerspiegeln könnte, fehlt und muss dementsprechend vollständig selbst übernommen werden. Das erscheint auf den ersten Blick als sehr großer Mangel der Doclet-Spezifikation, erklärt sich bei genauerer Betrachtung allerdings von selbst. Der Hauptgrund dafür ist, dass alle zum Standard-Doclet gehörenden Taglets nicht darauf ausgerichtet sind, andere Formate zu unterstützen. Auch wenn wir in unserem Beispiel ebenfalls HTML als Zielformat gewählt haben, sind die Taglets für die ausschließliche Verwendung durch das Standard-Doclet-API ausgelegt. Das wird auch dadurch transparent, dass deren Implementierung im Package *com.sun.tools.doclets.internal.toolkit.taglets* vorliegt, das somit architektonisch dem Standard-Doclet gehört. Eine Wiederverwendung für eigene Erzeugnisse wird dahingehend erschwert, dass diese Taglets ein eigenes Interface implementieren, das den Interfaces des Taglet-API nur auf den ersten Blick ähnelt. Wer also Javadoc mit einem eigenen Doclet erweitern möchte, muss sich zwangsläufig um die Tags, die in der Tabelle am Anfang dieses Artikels aufgelistet wurden, selbst kümmern. Dass es doch einen Weg gibt, der einem Hack gleicht, soll das das Thema Doclet abschließende Listing 10 zeigen, dessen Verwendung aus Listing 11 und dem Beispielprojekt [7] transparent wird.

Die Klasse übernimmt die Eigenschaften der Taglet-Implementierung des Standard-Doclets. Einzig das Rendering wird von einer eigenen Logik umgesetzt.

Happy Debugging

Mit den beiden APIs *Taglets* und *Doclets* liegen uns überschaubare Spezifikationen vor. Jedoch können damit erstellte Javadoc-Erweiterungen auch einmal an einen Punkt gelangen, der den Einsatz eines Debuggers erforderlich macht. Hierfür wäre es wünschenswert, den Javadoc-Interpreter in einem Debug-Modus zu starten. Doch leider wird man in diesem Punkt schnell enttäuscht, denn das Schlagwort „Debug" sucht man in den zur Verfügung stehenden Optionen vergebens. Dass es dennoch möglich ist, zeigt Listing 12, in dem eine *Main*-Klasse aus dem Package *com.sun.tools.javadoc* angesprochen wird, dem offiziellen Startpunkt für Javadoc und dem inoffiziellen Aufsetzpunkt für unseren Debugger. Über das von der *Main*-Methode erwartete String Array werden die Kommandozeilenparameter des Javadoc-Tools erwartet, sodass wir hier unsere eigene Implementierung ins Rennen schicken können, um es zu debuggen oder anderweitige Dinge zu initiieren.

Ein Blick in die Zukunft

Auch wenn wir mit den in diesem Artikel vorgestellten APIs scheinbar alle Möglichkeiten ausgeschöpft haben, die uns Javadoc bereitstellt, haben wir das Ende der Fahnenstange noch nicht erreicht. Der mit dem „JDK Enhancement Proposal" JEP172 [12] eingereichte Vorschlag greift das in Java 8 integrierte und standardmäßig aktive Doclint (JEP 105, [13]) auf und beschreibt ein *DocTree* [14] genanntes API, dass eine Abstraktion des Javadoc-Tools selbst darstellt, das sogar so weit geht, dass neben der bisher bekannten Standalone-Lösung auch eine Integration in den Compiler möglich wird. Letzterer Aspekt hat zum Ziel, die Aktualität der Dokumentation möglichst aktuell zu halten und Fehler in der selbigen frühzeitig, nämlich während des Compilerlaufs, zu erkennen. Ob und wie weit sich diese Idee durchsetzt, werden die nächsten Monate und Jahre zeigen.

Eines steht fest, *DocTree* wird Javadoc verändern und auf Dauer das als langsam bezeichnete Original gewiss um einige Alternativen bereichern.

Listing 12

```
@SuppressWarnings("restriction")
public class BaseDocletTester {
  public static void main(String[] args) {
    com.sun.tools.javadoc.Main.main("@options", "@params", "-doclet", "net.hofrichter...
                                                                    BaseDoclet");
  }
}
```

Fazit

Dass die Erstellung einer Dokumentation immer wieder ein leidiger Abschnitt bei der Erstellung von Software ist, ist allgemein bekannt und wird oft als Pflicht mit entsprechend reduziertem Spaßfaktor angesehen. Dennoch, oder gerade aus diesem Grund, hoffe ich, mit meinem kurzen Ausritt in die Welt der *Taglets* und *Doclets* einen interessanten Einblick in die Möglichkeit der Dokumentation gegeben zu haben. Wer gern eigene Javadoc-Erweiterungen verwenden möchte, für deren Umsetzung jedoch keine Muße oder Zeit hat, dem empfehle ich einen Blick auf die Javadoc-FAQs [15], auch wenn die dort verfügbare Liste an Drittanbietern nicht in allen Belangen korrekt ist und in einigen Fällen ins Leere läuft.

Sven Hofrichter ist Mitarbeiter der Finanz Informatik Solutions Plus GmbH, die sich auf Outsourcing-, Beratungs-, Entwicklungs- und Integrationsdienstleistungen für Geschäftsanwendungen in der Finanzwirtschaft spezialisiert hat. Der Autor ist als JEE-Architekt in Kundenprojekten tätig.

Links & Literatur

[1] http://docs.oracle.com/javase/8/docs/

[2] http://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html

[3] http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

[4] http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html#javadoctags

[5] http://www.oracle.com/technetwork/java/javase/tech/serializationfaq-jsp-136699.html

[6] http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serial-arch.html#6250

[7] https://github.com/hofrichter/doclets

[8] https://docs.oracle.com/javase/8/docs/technotes/tools/unix/javadoc.html#tag

[9] https://maven.apache.org/plugins/maven-javadoc-plugin/examples/taglet-configuration.html

[10] http://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/doclet/overview.html

[11] http://docs.oracle.com/javase/8/docs/jdk/api/javadoc/doclet/index.html

[12] http://www.doclet.com

[13] http://openjdk.java.net/jeps/172

[14] http://openjdk.java.net/jeps/105

[15] http://www.oracle.com/technetwork/java/javase/documentation/index-137483.html#docletsfromthirdparties

[16] https://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/standard-doclet.html

[17] https://docs.oracle.com/javase/8/docs/jdk/api/javac/tree/index.html