Annotation in Java 1.5

Metadaten

Oliver Szymanski

oliver.szymanski@mathema.de

www.mathema.de

MATHEMA

Themen

- Einstieg in Metadaten und Annotationen
- Zugriff auf Annotationen zur Laufzeit
- Zugriff auf Annotationen auf Quellcodeebene
- Standardannotationen
- Eigene Annotationen definieren

Metadaten

- Eine Vielzahl an Metadaten sind bei der Entwicklung erforderlich
  - z.B. soll eine Klasse als Webservice verfügbar sein
  - z.B. soll eine bestimmte Methode als UnitTest ausgeführt werden
  - Vgl. Deployment Deskriptoren in J2EE oder Xdoclet (Metadaten in Kommentaren)
- Benötigt wurde eine Möglichkeit diese Metadaten direkt in Java zu formulieren
  - Prüfung der Syntax / des Vorhandenseins durch den Compiler
  - Metadaten direkt an richtiger Stelle

Annotationen

- Metadaten können ab Java 1.5 als Annotationen formuliert werden
- Annotationen sind Erweiterungen der Java-Sprache
- Beispiel einer Annotation:

```
@Documented
public class LibraryContext {
   ...
}
```

- Im Beispiel wird der Klasse LibraryContext die Annotation „Documented“ zugeordnet. Diese kann u.a. durch das „ndoc“-Tool ausgewertet werden.

Parameter zu Annotationen

- Annotationen können auch Parameter bekommen:

```
import njunit.annotation.*;

public class Example {

   @UnitTest(value="This test will pass.")
   public void pass() {
      assert 10 > 5;
   }
```

- Im Beispiel wird eine Annotation aus dem UnitTest-Framework genutzt. Diese kennzeichnet Methoden, welche als UnitTest ausgeführt werden sollen, als Parameter wird die Beschreibung des UnitTests angegeben.

Parameter zu Annotationen

- Parameter sind Name/Wert Paare
- Mehrere Parameter werden durch Komma getrennt angegeben
- Die Reihenfolge bei Parametern ist egal
- Gibt es nur einen Parameter kann man auf den Parameternamen verzichten und nur den Wert angeben

Annotationen zur Laufzeit

- Man kann Annotationen von Typen per Reflection abfragen
- Dazu gibt es das Interface java.lang.reflect.AnnotatedElement
- Folgende Typen implementieren dieses Interface:
  - java.lang.reflect.AccessibleObject
  - java.lang.Class
  - java.lang.reflect.Constructor
  - java.lang.reflect.Field
  - java.lang.reflect.Method
  - java.lang.Package

Das Interface AnnotatedElement

- boolean isAnnotationPresent
  (Class<? extends Annotation> annotationType)
  - Liefert true wenn der als Parameter übergebene Typ als Annotation an dem Element vorhanden ist, sonst false
- <T extends Annotation> T getAnnotation
  (Class<T> annotationType)
  - Liefert die Annotation des übergebenen Typs, wenn zu dem Element eine vorliegt, sonst null
- Annotation[] getAnnotations()
  - Liefert alle zu dem Element vorliegende Annotationen
- Annotation[] getDeclaredAnnotations()
  - Liefert die direkt angegebenen Annotationen, nicht geerbte

Annotationen zur Laufzeit

- Bsp. die Klasse Testrunner, welche die UnitTest einer Klasse ausführt:

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

public class TestRunner {
   static void executeUnitTests(String className) {
      try {
         Object testObject = Class.forName(className).newInstance();
         Method [] methods = testObject.getClass().getDeclaredMethods();
         for(Method amethod : methods) {
         UnitTest utAnnotation = amethod.getAnnotation(UnitTest.class);
         if(utAnnotation!=null) {
            System.out.print(utAnnotation.value() + " : " );
            String result = invoke(amethod, testObject);
                 // invoke per Reflection hier nicht gezeigt
            System.out.println(result);
            }
         }
      }catch(Exception x) {
         x.printStackTrace();
      }
   }

   ...
}
```

Annotationen auf Quellcodeebene

- Quellcodeanalyse von Annotationen kann z.B. mit Hilfe der Doclet API implementiert werden:

```
import com.sun.javadoc.*;
public class ListAnnotations {
   public static boolean start(RootDoc root) {
      ClassDoc[] classes = root.classes();
      for (ClassDoc clsDoc : classes) { process(clsDoc); }
      return true;
   }
   static void process(ClassDoc clsDoc) {
      System.out.println("Annotations in " +
          clsDoc.name() + ":");
      AnnotationDesc[] annDescs = clsDoc.annotations();

      for (AnnotationDesc ad : annDescs) {
         AnnotationTypeDoc at = ad.annotationType();
         System.out.println("Annotation: " + at.name());
         AnnotationDesc.MemberValuePair [] members =
            ad.memberValues();
         for(AnnotationDesc.MemberValuePair mvp : members) {
            System.out.println("member = " +
                  mvp.member().name() +
                  ", value = "+ mvp.value() + "");
         }
      }
   }
}
```

Quellcodenanalyse von Annotationen

- Kompilieren
  - javac -source 1.5 -cp c:\jdk15\lib\tools.jar ListAnnotations.java
- Starten des Doclets
  - javadoc -source 1.5 -doclet ListAnnotations -sourcepath . UnitTest.java

Standardannotationen

- Es gibt sechs fertige Annotationen:
  - @Deprecated
    - Ersetzt den „deprecated“-javadoc-Tag. Compiler warnt, wenn ein Element mit dieser Annotation genutzt wird.
  - @Overrides
    - Wird bei Methoden genutzt, um anzugeben, dass man die Methode überschreiben möchten. Führt zu einem Compilerfehler, wenn man eine Methode irrtümlich nicht überschreibt sondern überlädt.
  - @Documented
    - Für Typen die in (Javadoc-) Dokumentationen aufgenommen werden sollen.

Standardannotationen

- @Inherited
  - Kann für eigenen Annotationen gesetzt werden, und bedeutet das diese vererbt werden, wenn sie zu einer Klasse gesetzt werden
- @Target
  - Kann für eigene Annotationen gesetzt werden. Gibt den Typ an, auf den eine Annotation angewendet werden darf. Z.B.: @Target({ElementType.FIELD, ElementType.METHOD}).Die möglichen Werte liegen in der Enumeration ElementType.
- @Retention
  - Gibt an, wann die Annotation zur Verfügung steht und ausgewertet werden kann. Möglich ist Auswertung nur im Quellcode, vom Compiler, zur Laufzeit.

Beispiele zu den Standardannotationen

- @Deprecated

```
public class Parent {
   @Deprecated
   public void foo(int x) {
      System.out.println("This method is deprecated and should not be used.");
   }
}
```

- @Overrides

```
class A {
        public void f(A a) { ... }
}
class B extends A {
        public void f(B a) { ... }  // gewünscht war eigentlich f(A a)
                                    // zu überschreiben nicht zu überladen
}
class C extends A {
        @Overrides public void f(A a) { ... } // überschreiben: OK
        @Overrides public void f(B a) { ... } // überladen: ERROR
}
```

Eigene Annotation erstellen

- Annotationen werden ähnlich einem Interface mit @interface definiert
  - extends darf nicht genutzt werden (Annotationen erweitern automatisch das neue Marker-Interface java.lang.annotation.Annotation)
  - Angegebene Methoden haben keine Parameter
  - Methoden haben keine Typ-Parameters (keine generischen Methoden)
  - Methodrückgabetypen sind beschränkt auf primitive Type, String, Class, enum-Typen, Annotation-typen, und Arrays von diesen
  - Throws darf nicht deklariert werden
  - Annotationstypen dürfen nicht parametrisiert werden

Eigene Annotationen erstellen

- Einfache Annotation zur Markierung

```
public @interface Preliminary { }
```

- Einfache Annotation primitiven Werttypen

```
public @interface RequestForEnhancement {
   int    id();              // Unique ID number associated with RFE
   String synopsis();        // Synopsis of RFE
   String engineer();        // Name of implementor
   String date();            // Date RFE was implemented
}
```

```
public @interface Authors {
   String[] names();
}
```

- Array werden dann als Parameter wie folgt angegeben:

```
@Authors({"xyz", "abc"})
```

  - Soll nur ein Wert übergebenen werden, kann man die geschweiften Klammern wegfallen lassen

Eigene Annotationen erstellen

- Komplexe Annotation

```
public @interface Name {
    String first();
    String last();
}
public @interface Author {
    Name value();
}
public @interface Reviewer {
    Name value();
}
```

  - werden wie folgt genutzt:

```
@Author(@Name(first = "Joe", last = "Hacker"))
```

Eigene Annotationen erstellen

- Typ-gebundene Annotationen

```
public interface Formatter { ... }

/**
 * Designates a formatter to pretty-print the annotated class.
 */
public @interface PrettyPrinter {
    Class<? extends Formatter> value();
}
```

- Default-Werte können definiert werden, so dass die Angabe des Parameters bei Nutzung der Annotation nicht mehr nötig ist:

```
public @interface RequestForEnhancement {
    int    id();              // No default
    String synopsis();        // No default
    String engineer()         default "[unassigned]";
    String date()             default "[unimplemented]";
}
```

Defaultwerte

- Annotationen können problemlos um Attribute mit Default-Werten erweitert werden,
  - ansonsten: IncompleteAnnotationError
- Entfernen eines Attributs führt zu einem NoSuchMethodError
- Ändern eines Attribut-Typs führt ggf. zu einem AnnotationTypeMismatchError
- Entfernen von Defaults führt ggf. zu einem IncompleteAnnotationError
- Ein ändern der Retention-Politik oder von Targets kann zu Problemen zur Übersetzungszeit und oder Laufzeit führen

Beispiele

- Source-Level-Annotation:

```
@Retention(SOURCE)
public @interface Copyright {
   String text();
}
```

  - Werte für Retention kommen aus RetentionPolicy. Möglich ist:
    - SOURCE: Annotationen landen nicht im Class-File
    - CLASS: Annotationen landen im Class-File, werden aber nicht zur Laufzeit ausgewertet (Default, wenn Retention nicht angegeben wird)
    - RUNTIME: Annotation landen im Class-File und können zur Laufzeit per Reflection ausgewertet werden

Beispiele

- Definition der Inherited-Annotation:

```
@Documented @Retention(RUNTIME) @Target(ANNOTATION_TYPE)
public @interface Inherited {
}
```

- Definition der Retention-Annotation:

```
@Documented @Retention(RUNTIME) @Target(ANNOTATION_TYPE)
public @interface Retention {
   RetentionPolicy value();
}
```

- Definition der Target-Annotation:

```
@Documented @Retention(RUNTIME) @Target(ANNOTATION_TYPE)
public @interface Target {
   ElementType[] value;
}
```

Vielen Dank!

oliver.szymanski@mathema.de

www.mathema.de

MATHEMA