Programmieren 3

Kapitel 8: Annotationen

- Annotationen - Motivation und Verwendung
- Definition eigener Annotationen
- Wichtige Anwendungen von Annotationen (JUnit 4 etc)

Worum geht es?

- Was sind Annotationen?
- Wie verwende ich Annotationen?
- Wie definiere ich eigene Annotationen?
- Welche Annotationen gibt es bereits?

Motivation

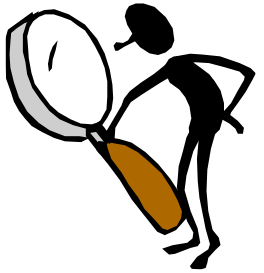
- In Java 5 wurden als neues Sprachkonzept Annotationen eingeführt.
- Annotationen = „Bemerkungen“
  - sind (im Quelltext des Programmes aufgeführte) Metainformationen.
  - stellen zusätzliche semantische Informationen zum Programm bereit.

Annotationen sind Meta-Informationen (d.h. Informationen über Informationen).

- Semantik von Annotationen
  - Semantik von Annotationen wird vom Compiler nicht „direkt“ bearbeitet, d.h. haben keinen direkten Einfluss auf die Semantik des Elements, bei dem sie aufgeführt werden
  - Beeinflussen jedoch die Semantik von „Dingen“, die diese Elemente verwenden

Verwendung von Annotationen

- Annotationen können zu verschiedenen Zwecken verwendet werden:
  - Informationen für den **Compiler**
    Annotationen können z.B. vom Compiler verwendet werden, um Fehler zu erkennen, Warnungen zu generieren
  - Informationen für **Werkzeuge** (Tools)
    Tools können vor dem Übersetzen (Compile-time) oder zum Deployment (Deployment-time) Annotationen abfragen, um Java Quellcode, XML-Dateien oder anderes zu erzeugen (Beispiele: Junit 4, Hibernate)



  - Informationen zur **Laufzeit**
    Gewisse Annotationen stehen auch zur Laufzeit zur Abfrage über ein entsprechendes API zur Verfügung (→ Reflection!).
- Im Programmieralltag:
  Annotationen **häufig verwenden**, jedoch **selten** neue **definieren**!

Wichtige Vordefinierte Annotationen (1)

- `@Override`
  - Typischer Programmierfehler: man will eine Methode überschreiben, überlädt sie jedoch statt dessen.
    - Beispiel:

```
// falsch und nicht entdeckt
public boolean equals(Rational other) { ... }

// richtig
public boolean equals(Object other) { ... }
```

  - Durch annotieren der Methode mittels `@Override` wird dieser Fehler frühzeitig entdeckt (Compilerfehler)
    - Beispiel:

```
// falsch und sofort entdeckt!
@Override
public boolean equals(Rational other) { ... } // compilation error

// richtig
@Override
public boolean equals(Object other) { ... } // ok
```

Wichtige Vordefinierte Annotationen (2)

- `@SuppressWarnings`
  - Aktuelle Java-Compiler erzeugen sehr viele Warnungen
  - Viele davon sind berechtigt und weisen auf Programmierfehler hin
  - Einige sind jedoch „falsch“, d.h. der Code ist so gewollt → Wunsch, Warnungen abschalten & gleichzeitig dokumentieren, dass man sich sicher ist
  - Art der zu unterdrückenden Warnung ist Parameter (und Compiler-spezifisch)
  - Beispiel für Eclipse:

```
....

@SuppressWarnings("unchecked")
public void methodWithScaryWarning() {
    List rawList = new ArrayList();
    List<String> stringList = (List<String>)rawList;
}

....
```

Wichtige Vordefinierte Annotationen (3)

- `@Deprecated`
  - Hinweis an den Programmierer, diese Methode/Klasse/etc. nicht (mehr) zu verwenden
  - Mögliche Gründe:
    - Fehler in bestimmten Fällen
    - Bessere (neuere) Alternative existiert
  - Compiler erzeugt eine Warnung, wenn Elemente die als `@deprecated` annotiert sind, benutzt werden
  - Beispiel:

```
class AlteKlasse {
    @Deprecated
    public static void doofeAlteMethode() {
        ....
    }
}

class NeueKlasse {
    public boolean meineNeueMethode() {
        doofeAlteMethode();  // Compiler Warning
    }
}
```

Definition von neuen Annotationen

- Annotationen werden ähnlich Interfaces definiert, jedoch mit dem @-Zeichen vor dem Schlüsselwort Interface
- In der Definition einer Annotation können Methoden deklariert werden, die Elemente der Annotation beschreiben.
  - Methoden in Annotationen besitzen keine Parameter.
  - Erlaubte Rückgabetypen: byte, short, int, long, float, double, String, Class, Annotation und Enumeration sowie Felder über diese Typen
  - Definition von Defaultwerten (von den Rückgabetyp) möglich.
- Beispiel: Annotation `@Fixed` um anzuzeigen, wer wann welchen Fehler zu beheben versucht hat.

```
public @interface Fixed {
    String author() ;
    String date() ;
    String bugsFixed() default "" ;
}
```

Verwendung eigener Annotationen

- Beispiel

```
@Fixed(author="M. Breunig", date="29.12.2010")
public void tolleFunktion() {
    ...
}
```

- Beachte
  - Für die dritte Methode `bugsFixed` in der Annotation ist kein Wert angegeben.
  - Hier wird der Default-Wert "" verwendet.
- Annotationen können an beliebige Typdeklarationen (Objekt-/Klasse-/lokale Variablen, Parameter), Methoden oder Konstruktoren hinzugefügt werden.

Beispiel Annotation @Column

- `@Column` legt für jedes Attribut einer Klasse die Spalte zu einer Tabelle fest
- Definition der Annotation

```
public @interface Column {
    String name();
    String type() default = “VARCHAR2”;
}
```

- Verwendung der Annotation

```
@Table(name = "CustomerTable")
public class Kunde {

   @Column(name="nr", type="VARCHAR2(10)")
   private String nummer;
   @Column(name="vorname")
   private String vorname;
```

Schreibvereinfachungen für Annotationen

- Annotationen ohne Methoden: Marker-Annotationen
  - Runde Klammern können entfallen
  - Beispiel: `@Override, @Deprecated`
- Annotationen mit genau einer Methode: Value-Annotationen
  - Nur eine einzige Methode mit Namen `value`
  - Bezeichner `value` kann bei Verwendung weggelassen werden
  - Beispiel

```
public @interface ReleaseVersion {
    String value() ;
}


@ReleaseVersion("1.2.5")
public class Calculator {
    ...
}
```

Vordefinierte Meta-Annotationen

- Meta-Annotationen: Annotieren neu-definierte Annotationen
- `@Target`
  - Legt fest, zu welcher Art von Java-Element die Annotationen gehört
  - Beispiel:
```
@Target({ElementType.FIELD, ElementType.METHOD})
public @interface Fixed{ … }
```
- `@Retention`
  - Legt fest, ob eine Annotation nur im Source-Code vorhanden ist, oder auch im Binärcode (default-Fall)
  - Beispiele:
```
@Retention(RetentionPolicy.RUNTIME),
@Retention(RetentionPolicy.CLASS),
@Retention(RetentionPolicy.SOURCE)
```
- `@Inherited`
  - Legt fest, dass die Annotation an Unterklasse vererbt wird
  - Beispiel: `@Inherited`

Meta-Annotationen
sind Meta-Informationen über Meta-Informationen

(d.h. Informationen über Informationen über Informationen).

Auswerten von Annotationen

Das JDK enthält zwei Tools die das Auswerten von Annotationen ermöglichen:

1) Annotation Processing Tool (APT)
   - ermöglicht das Auslesen von Annotationen aus dem Quell- oder dem Bytecode.

2) Reflection-API
   - erlaubt das Abfragen von Annotationen zur Laufzeit eines Java-Programms.
   - Interface `AnnotatedElement`
     - Von den Reflection-Klassen `Class`, `Method`, `Field` etc implementiert
     - Enthält Methoden zum Zugriff auf die Annotationen (wie `getAnnotations` oder `isAnnotationPresent` etc.)

Auslesen der Annotation zur Laufzeit

```
public static void print(Class<? extends Object> cls) {
   Table table = (Table) cls.getAnnotation(Table.class);
   System.out.println("Tabelle = " + table.name());

   Field fields[] =  cls.getDeclaredFields();
   for (Field field : fields) {
     Column column = (Column) field.getAnnotation(Column.class);
     System.out.println("  " + column.name() +
         "(" + column.type() + ")");
   }
}
```

- Nutzung Java-Reflection API für @Table und @Column
- Auslesen der Attribute

Anwendungsbeispiele

Beispiele für Anwendungen des neuen Annotationsmechanismus

- Java Enterprise Beans (EJB)
  - Viele notwendige Informationen nicht mehr in separaten Konfigurationsdateien im XML-Format, sondern durch Annotationen spezifiziert.

- Objekt-relationale Abbildung (JPA, Hibernate)
  - Alle notwendige Zusatzinformation zum Generieren entsprechender Wertklassen und der notwendigen Datenbankzugriffe über Annotationen spezifiziert (alternativ: XML-Konfigurationsdateien)

HIBERNATE

- JUnit 4
  - Testframework – Annotation der Testcases

JUnit.org Resources for Test Driven Development

JUnit 4

- JUnit 4 ist die (erhebliche) Weiterentwicklung von JUnit 3
- Statt Methodennamen `testXXX` werden Annotationen zur Markierung der Testcases verwendet → annotation based testing
- Änderungen gegenüber JUnit 3
  - Nicht mehr von `TestCase` erben
  - Methoden müssen nicht mehr mit `test`… beginnen (sollten aber)
  - Statt dessen mit `@Test` annotieren
  - Statt `setup` und `tearDown`, Methoden die vor/nach jedem Testcase laufen sollen mit `@Before` und `@After` annotieren
  - Methoden, die einmalig vor/nach allen Testcases laufen sollen mit `@BeforeClass` und `@AfterClass` annotieren, diese Methoden müssen statisch sein

Junit 4 Beispiel

```
public class SimpleClassTest {
  private SimpleClass sc;
  private static Logger logger = Logger.getLogger(SimpleClassTest.class.getName());

  @BeforeClass
  public static void setUpBeforeClass() {
    logger.info("Starte Test");  }
  @AfterClass
  public static void tearDownAfterClass() {
    logger.info("Test Fertig");  }
  @Before
  public void setUp() throws Exception {
    logger.info("Init Case");
    sc = new SimpleClass();  }
  @After
  public void tearDown() throws Exception {
    logger.info("Cleanup Case");
    sc = null;  }
  @Test
  public void testAdd() {
    logger.info("testing add");
    assertEquals(13, sc.add(8,5));  }
  @Test
  public void testSubtract() {
    logger.info("testing subtract");
    assertEquals(10, sc.subtract(20, 10));
  }
}
```

```
public class SimpleClass {
  public int add(int a, int b) {
    return a+b;
  }

  public int subtract(int a, int b) {
    return a-b;
  }
}
```

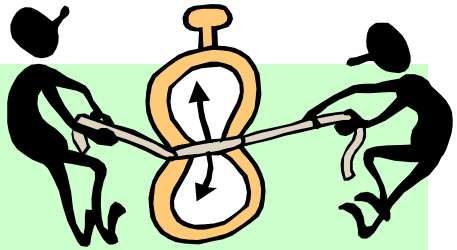
JUnit 4 – Fortgeschrittenes (1)

- `@Ignore`: (Temporäres) Abschalten von Tests.
  - Optionaler String-Parameter mit dem Grund
  - Beispiel

```
@Ignore("Not Ready to Run")
@Test
public void multiplicationstest() {
  assertEquals(15, sc.multiply(3, 5));
}
```

- `timeout`-Parameter
  - Test „failed“ automatisch, falls er länger als die angegebenen Millisekunden läuft
  - Beispiel

```
@Test(timeout = 1000)
public void testSubtract() {
  logger.info("testing subtract");
  assertEquals(10, sc.subtract(20, 10));
}
```



JUnit 4 – Fortgeschrittenes (2)

- Testen von Exceptions – `expected`-Parameter
  - Erwartete Exception angeben
  - Beispiel:
    bisher (in JUnit 3)

```
  public void testDivideWithException() {
    try {
      sc.divide(3,0); // sollte ArithmeticException werfen
      fail();
    } catch(ArithmeticException ae) {
    }
}
```

  - Nun (in JUnit 4)

```
  @Test(expected = ArithmeticException.class)
  public void testDivideWithException() {
    sc.divide(3,0)
  }
```

Verarbeitung von Annotationen durch Tools

- Beispiel: XML Schema Generator aus JAXB
- Aufruf von: **`schemagen Kunde.java`**

```
@XmlRootElement(name="KundeRoot")
@XmlAccessorType(XmlAccessType.NONE)
@XmlType(name=„Customer")
public class Kunde {

@XmlAttribute(name="number", required=true)
private String nummer;
@XmlElement(name="fullname", required=false)
    private String name;
```

Generiert aus Java-Klasse eine XML-Schema Datei

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="KundeRoot" type="Customer"/>
  <xs:complexType name=„Customer">
    <xs:sequence>
      <xs:element name="fullname" type="xs:string" />
      <xs:attribute name="number" type="xs:string" use="required"/>
```

Beispiel: Annotationen bei Java Webservices

- Soap Webservice

```
@WebService(targetNamespace = "http://duke.example.org", name="AddNumbers")
public interface AddNumbersIF {
    @WebMethod(operationName="add", action="urn:addNumbers")
    @WebResult(name="return")
    public int addNumbers(@WebParam(name="num1")int number1,
                          @WebParam(name="num2")int number2)
                          throws AddNumbersException;
}
```

- REST Webservice

```
@Path("/customerservice/")
@Produces("application/xml"); // Alternativ: @Produces("application/json");
public interface ICustomerService {
   @GET
   @Path("/customers/{id}/")
   public Customer getCustomer(@PathParam("id") String id);
   @POST
   @Path("/customers/")
   public Response addCustomer(Customer customer);
}
```

- Frameworks, Tools oder Applicationserver erzeugen aus Annotationen Service-Schnittstellenbeschreibung und Proxy-Klassen für Serviceaufruf

Zusammenfassung Annotationen

- Annotationen ermöglichen Metainformationen in Programmcode einzufügen
- Annotationen sind ein Schritt in die deklarative Programmierung
- Annotationen sind sehr stark mit dem Quellcode verbunden und können nur dort geändert werden
- Klassen mit Annotationen sind invasiv, d.h. sie binden die Implementierungen der Annotationen an den Klassenpfad
- Annotationen werden häufig von Tools ausgewertet
- JUnit 4 verwendet Annotationen