Programmieren - Exkurse

Matthias Berg-Neels

Download Skript

Inhalt

- Naming conventions
- Implizite Typisierung mittels `var`
- Unit Tests
- Innere Klassen
- Entwurfsmuster (Design Patterns)
- (Die Evolution von Java)
- (Optionals)
- (Programming Principals)

Naming conventions

Naming conventions sind, in (großen) Projekten / Firmen, Bestandteil der "Code Style Guidelines". Firmen bzw. Community basiert gibt es unterschiedliche Code Style Guidelines nach denen man sich richten kann. Die folgenden Konventionen basieren auf den Google Style Guidelines für Java.

Allgemeine Regeln für Bezeichner und Namen (1/2)

Regeln

- erlaubte Zeichen
  - Buchstaben (Case sensitive): a, b, c, ..., x, y, z, A, B, ..., Y, Z
    - Landespezifische Zeichen sind erlaubt (z.B. ä, ü, ö) - sollten aber vermieden werden! (Stichwort: Kompatibilität zwischen Rechnern - betrifft NUR Quellcode, nicht den Bytecode!)
  - Unterstrich: _
  - Dollarzeichen: $
  - Zahlen: 0, 1, 2, ..., 9
- nicht erlaubte Zeichensatz
  - Sonderzeichen

Allgemeine Regeln für Bezeichner und Namen (2/2)

weitere Regeln

- keine Leerzeichen
- dürfen nicht mit Zahlen beginnen
- dürfen nicht gleich mit reservierten Schlüsselwörtern sein

Empfehlungen

- sprechende Namen -> man kann beim Lesen verstehen was eine Variable, Klasse, Methode für eine Aufgabe hat.
- **Merksatz**: Wenn man einen Namen lesen kann ohne "grammatikalische" Bauchschmerzen zu bekommen, ist es die richtige Richtung.
- **Anmerkung**: Namen sollten immer die tatsächliche Funktion einer Entität widerspiegeln, daher stehen sie in hoher Abhängigkeit zum Quellcode. Eine Variable mit dem Namen "WindowCount" (vom Typ Boolean), eine Methode "saveToDatabase" (die nichts Speichert) oder eine Klasse "Student" (die Funktionen einer Vorlesung enthält) sollten namentlich noch einmal überdacht werden, auch wenn diese sich "rein vom Namen" korrekt lesen.

Packagenamen

Guideline

- klein geschrieben

```
de.mbn.myapp.lecture
lecture.objectorientation.trainstation
```

Klassennamen

Guideline

- beginnen mit einem Großbuchstaben
- UpperCamelCase - beginnen mit Großbuchstaben und jedes neue Wort beginnt mit einem Großbuchstaben
- Zusatz: Der Dateiname muss sich nach dem Namen der Hauptklasse (first level) in der Datei richten

```
Car
Student
TrainDriver
```

Variablen / Attribute

Guideline

- lowerCamelCase - beginnen mit einem Kleinbuchstaben und jedes neue Wort beginnt mit einem Großbuchstaben

```
familyName
children
studentId
```

Konstanten

Guideline

- UPPER_CASE - werden vollständig mit Großbuchstaben geschrieben, neue Wörter werden durch Unterstrich getrennt

```
ALLOWED_COLOR_RED
MEANING_OF_LIFE
```

Methoden

Guideline

- lowerCamelCase - beginnen mit einem Kleinbuchstaben und jedes neue Wort beginnt mit einem Großbuchstaben
- beginnen mit einem Verb (bzw enthalten mindestens ein Verb)
  - eine Methode "spiegelt" eine Tätigkeit, Geschehen, Vorgang wieder --> es passiert etwas

```
accelerate();
persistData();
```

Getter- / Setter

- Attributname wird mit get bzw. set vorangestellt in lowerCamelCase

```
setFamilyName();
getFamilyName();
```

Spezialfall: Boolen Attribute / Getter-Methoden

- sprechende Definition von Boolean-Attribute
  - z.B. enabled, isTired (VS tired), hasFlatRoof (VS flatRoof), canFly (VS fly), ...
- Boolean Getter-Methoden werden nicht mit get, sondern mit dem passenden Verb (is, has, can) gebildet
  - isEnabled(), isTired(), hasFlatRoof(), canFly()
- anhängig vom Attributnamen können in diesem Fall die Setter-Namen doch komisch wirken
  - setEnabled, setIsTired (VS setTired), setHasFlatRoof (VS setFlatRoof), setCanFly (VS setFly)

Ein Beispiel aus dem echten Leben (/ produktiven Code)

```
package com.sap.iot.rules.ruleprocessorstream.cache.repository;

import // ...

public class ThingModelBasedDataObjectRepository extends HashOperationsRepository<ThingModelBasedDataObject> {

    private static final String KEY_PREFIX = "do_by_rsid_tt_ps";

    private static final String NAME = "name";
    private static final String TYPE = "type";
    private static final String IS_RESULT = "isResult";
    private static final String THING_TYPE = "thingType";
    private static final String PROPERTY_SET = "propertySet";
    private static final String PROPERTY_SET_TYPE = "propertySetType";
    private static final String SENSITIVITY_LEVEL = "sensitivityLevel";
    private static final String LIST_SIZE_SUFFIX = "size";
    private static final String USED_ATTRIBUTES_PREFIX = "usedAttributes.";
    private static final String WILDCARD = "*";

    public ThingModelBasedDataObjectRepository(@Qualifier("ruleCache") RedisTemplate<String, String> redisTemplate, TenantContext tenantContext) {
        // ...
    }

    @Override
    public String getUniqueCachingKeyPrefix() {
        // ...
    }

    @Override
    public void save(@NotBlank String ermRuleServiceId, @Valid ThingModelBasedDataObject dataObject) {
        // ...
    }

    public Optional<ThingModelBasedDataObject> find(@NotBlank String ermRuleServiceId, String thingType, String propertySet) {
        // ...
    }

    public List<ThingModelBasedDataObject> findAll(@NotBlank String ermRuleServiceId, String thingType) {
        // ...
    }

    public Optional<ThingModelBasedDataObject> find(@NotBlank String ermRuleServiceId, String propertySetType) {
        // ...
    }

    public Boolean delete(@NotBlank String ermRuleServiceId, String thingType, String propertySet, String propertySetType) {
        // ...
    }
```

implizite Typisierung mittels `var`

Ermittlung des Datentyps einer Variable ohne spezifische Angabe des Typs mittels der Initialisierung - Schlüsselwort `var` *(seit Java 10)* *some evil stuff*

Verwendung

Voraussetzung

- Deklaration mit `var` UND sofortiger Initialisierung der Variable.

Beispiele

```
var numberA = 10;
       // numberA wird zu Integer Variable deklariert
var numberB = 42.0;
     // numberB wird zu Double Variable deklariert
var textA = "Herzlich Willkommen";    // textA wird zu String Variable deklariert
var myAnimal = new Dog(...);          // myAnimal wird zu Dog Variable deklariert

var test;
               // Compiler Fehler!

int numberC = 100;

var numberD = numberC;
  // numberD wird zu Integer Variable deklariert
```

Besser lesbarer (kürzerer) Code durch Vermeidung von Redundanzen

```
// vorher:
ThingModelBasedDataObjectRepository myThingModelRepo =
      new ThingModelBasedDataObjectRepository(myChacheTemplate, customerTenant);

// neu:
var myThingModelRepo = new ThingModelBasedDataObjectRepository(myChacheTemplate, customerTenant);
```

ABER...

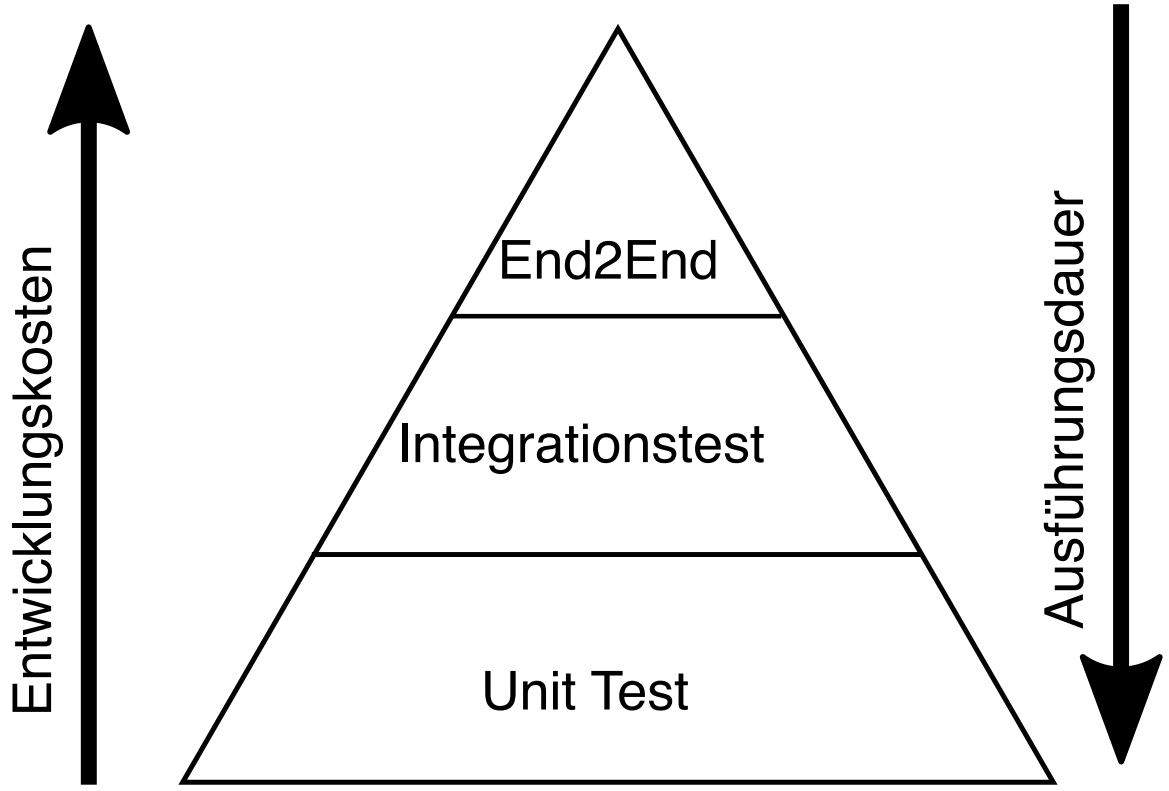
Falsch eingesetzt, wird der Code unverständlicher / komplizierter zu lesen:

```
var somethingOne = (farm.hasAnimal()) ? new Dog(...) : "Kein Tier";
var somethingTwo = ("Ergebnis ist " + (numberA + numberB * 50.1)).length() / (double)10;
var somethingThree = Something.returnSomething();
// ...
```

Unit Testing

Unit Testing in Java mit JUnit5.

Einordnung von Unit Tests



JUnit5 - Basis Annotationen

Annotation	Beschreibung
`@Test`	kennzeichnet eine Methode als Test
`@Tag("<tag>")`	definiert einen Tag zur Filterung von Tests
`@BeforeEach`	kennzeichnet eine Methode die **vor jedem** Test läuft (JUnit4 --> `@Before`)
`@AfterEach`	kennzeichnet eine Methode die **nach jedem** Test läuft (JUnit4 --> `@After`)
`@BeforeAll`	kennzeichnet eine Methode die **einmal vor allen** Tests läuft (JUnit4 --> `@BeforeClass`)
`@AfterAll`	kennzeichnet eine Methode die **einmal nach allen** Tests läuft (JUnit4 --> `@AfterClass`)

Assertion

- Zusicherung / Sicherstellung / Assertion (lat. Aussage / Behauptung)
  - Definition einer Erwartungshaltung zum Vergleich gegen den tatsächlichen Zustand
- JUnit Tests:
  - Klasse: `Assertions` (`org.junit.jupiter.api.Assertions`)
  - statische Methoden zur Definition eines erwartenden Ergebnisses (expected) zum Vergleich mit dem tatsächlichen Ergebnis (actual)
  - überladene Methoden mit zusätzlichem Parameter `Message` für eigene Meldungen
  - automatische Validierung der "Behauptung" durch das JUnit Test-Framework
  - beliebt als statischer Import zur direkten Nutzung der Methoden:

```
import static org.junit.jupiter.api.Assertions.*;
```

  - Beispiele:

```
Assertions.assertEquals(<expected>, <actual>[, <Message>]);
Assertions.assertNotEquals(<expected>, <actual>[, <Message>]);
Assertions.assertTrue(<actual>[, <Message>]);
Assertions.assertFalse(<actual>[, <Message>]);
Assertions.assertTimeout(<expected Duration>, <Executable>[, <Message>]);
Assertions.assertThrows(<expected Exception-Class>, <Executable>[, <Message>]);
```

JUnit5 - neue Annotationen

Annotation	Beschreibung
`@DisplayName("descriptive name")`	definiert den Anzeigename für den jeweiligen Test / die Testklasse
`@Nested`	markiert eine innere (geschachtelte) Testklasse -> Strukturierung
`@RepeatedTest(count)`	sich wiederholender Testfall
`@ParameterizedTest`	Testfall Parametrisierung -> siehe nächste Slide

`@ParameterizedTest`

- Separierung von Test-Code und Testfall
- verschiedene Quellen für Testfälle
  - `@ValueSource`
  - `@EmptySource` / `@NullSource` / `@NullAndEmptySource`
  - `@EnumSource`
  - `@CsvSource` / `@CsvFileSource`
  - `@MethodSource`

JUnit5 - Nützliches

Testen von Ausnahmen

```
Exception assertThrows(ExceptionClass, Executable)
```

Testen von mehrer Annotationen auf einmal

```
void assertAll(Executable ...);
```

Testen der Laufzeit

```
void assertTimeout(Duration, Executable);
```

Beispiel: Einfache Test-Klasse

```java
import excercises.exkurs.junit.Calculator;
import org.junit.jupiter.api.*;

class CalculatorTest {

    Calculator myCalculator;
    double result = 0;

    @BeforeEach
    void setUp() {
        myCalculator = new Calculator();
        result = 0;
    }

    @Test
    @DisplayName("adding two numbers")
    void add() {
        result = myCalculator.add(5.0, 10.0);
        Assertions.assertEquals(15.0, result);
    }
}
```

F.I.R.S.T. Principal

- **Fast**: Die Testausführung soll schnell sein, damit man sie möglichst oft ausführen kann. Je öfter man die Tests ausführt, desto schneller bemerkt man Fehler und desto einfacher ist es, diese zu beheben.
- **Independent**: Unit-Tests sind unabhängig voneinander, damit man sie in beliebiger Reihenfolge, parallel oder einzeln ausführen kann.
- **Repeatable**: Führt man einen Unit-Test mehrfach aus, muss er immer das gleiche Ergebnis liefern.
- **Self-Validating**: Ein Unit-Test soll entweder fehlschlagen oder gut gehen. Diese Entscheidung muss der Test treffen und als Ergebnis liefern. Es dürfen keine manuellen Prüfungen nötig sein.
- **Timely**: Man soll Unit-Tests vor der Entwicklung des Produktivcodes schreiben.

Innere Klassen

von inneren Klassen hin zu Lambda-Funktionen

Arten von inneren Klassen

- Innere Top-Level Klasse
  - Geschachtelte statische Klasse innerhalb einer anderen Klasse mit Bezeichner (Klassenname)
  - können innerhalb und außerhalb (abhängig von der Sichtbarkeit) der Klasse verwendet werden
- Innere Element Klasse
  - Geschachtelte Klasse innerhalb einer anderen Klasse mit Bezeichner (Klassenname)
  - können innerhalb und außerhalb (abhängig von der Sichtbarkeit) der Klasse verwendet werden
    - nur im Kontext eines Objekts der äußeren Klasse
- Innere lokale Klasse
  - Geschachtelte Klasse innerhalb einer Methode mit Bezeichner (Klassenname)
  - können nur innerhalb der Methode (Scope) genutzt werden
- Innere anonyme Klasse
  - geschachtelte Klasse innerhalb einer anderen Klasse / Methode **ohne** Bezeichner
  - werden direkt einer Referenz zugwiesen
  - basieren immer auf einer Klasse (erweitern) oder einem Interface (implementieren)

Innere Top-Level-Klasse

```java
package main.inner.toplevelclass;

public class OuterClass {

    // Innerhalnb einer anderen Klasse definierte Top-Level Klasse
    public static class InnerTopLevelClass{
        void print(String printText){
            System.out.println(this.getClass().getName() + " " + printText);
        }
    }

    private static void printFromInnerTopLevelClass(String printText) {
        OuterClass.InnerTopLevelClass myInnerTopLevelClass = new OuterClass.InnerTopLevelClass();
        myInnerTopLevelClass.print(printText);
    }

    public static void main(String[] args) {
        OuterClass myClass = new OuterClass();

        System.out.println("OuterClass: " + myClass.getClass().getName());
        OuterClass.printFromInnerTopLevelClass("Inner Top-Level Class: HelloWorld");
    }
}
```

Innere Element-Klasse

```
package main.inner.elementclass;

public class OuterClass {

    // Innerhalb einer andere Klasse definierte Element Klasse
    public class InnerElementClass {
        void print(String printText){
            System.out.println(this.getClass().getName() + " " + printText);
        }
    }

    void printFromInnerElementClass(String printText){
        OuterClass.InnerElementClass myInnerElementClass = this.new InnerElementClass();

        myInnerElementClass.print(printText);
    }

    public static void main(String[] args) {
        OuterClass myClass = new OuterClass();

        System.out.println("OuterClass: " + myClass.getClass().getName());
        myClass.printFromInnerElementClass("Inner Element Class: HelloWorld");
    }
}
```

Innere lokale Klasse

```
package main.inner.local;

public class OuterClass {

    void printFromLocalInnerClass(String printText){
        // innerhalb einer Methode (Scope) definierte Klasse
        class LocalInnerClass{
            void print(String printText){
  System.out.println(this.getClass().getName() + " " + printText);
            }
        }

        LocalInnerClass myLocalInnerClass = new LocalInnerClass();

        myLocalInnerClass.print(printText);
    }

    public static void main(String[] args) {
        OuterClass myClass = new OuterClass();

        System.out.println("OuterClass: " + myClass.getClass().getName());
        myClass.printFromLocalInnerClass("local inner Class: HelloWorld");
    }
}
```

Innere anonyme Klasse

```java
package main.inner.anonym;

public class OuterClass {

    private static interface Printable{
        void print(String printText);
    }

    void printFromAnonymousInnerClass(String printText) {
        // ohne eigenen Bezeichner definiert (kann nicht wiederverwendet werden)
        // erweitert eine bestehende Klasse oder implementiert ein Interface
        OuterClass.Printable myAnonymousInnerClass = new OuterClass.Printable() {
            @Override
            public void print(String printText) {
  System.out.println(this.getClass().getName() + " " + printText);
            }
        };

        myAnonymousInnerClass.print(printText);
    }

    public static void main(String[] args) {
        OuterClass myClass = new OuterClass();

        System.out.println("OuterClass: " + myClass.getClass().getName());
        myClass.printFromAnonymousInnerClass("Inner anonymous Class: HelloWorld");
    }
}
```

Lambda Funktionen (anonyme Funktionen)

- seit Java 8
- reine Funktionen ohne eigene Klasse
- Definition: ( ) –>{ }
- implementieren ein funktionales Interface (Interface mit **einer** Methode, **ohne default** Implementierung)
  - ersetzen (unter dieser Voraussetzung) anonyme Klassen
- haben Zugriff auf den umliegenden Kontext (finale / effektiv finale Variablen)
  - in diesem Zusammenhang auch als "Closure" bezeichnet
- verkürzte Schreibweise durch Herleitung der Informationen aus Interface-Definition
- werden an eine Referenz übergeben (direkt oder indirekt)

```
Interface1 lambda1 = parameter -> Anweisung;
Interface2 lambda2 = (parameter1, parameter2) -> Anweisung;
Interface3 lambda3 = () -> {
  Anweisung1;
  Anweisung2;
  Anweisung3;
}
```

Lambda Funktion

```java
package main.lambda;

public class OuterClass {

    private static interface Printable{
        void print(String printText);
    }

    void printFromLambdaFunction(String printText) {
        // Lambda Funktionen sind "reine Funktionen" ohne Klasse
        // nutzen immer ein funktionales Interface (nur eine Methode)
        // zur Implementierung
        OuterClass.Printable myLambdaPrintFunction = (lambdaPrintText) -> {
            System.out.println(this.getClass().getName() + " " + lambdaPrintText);
        };

        myLambdaPrintFunction.print(printText);
    }

    public static void main(String[] args) {
        OuterClass myClass = new OuterClass();

        System.out.println("OuterClass: " + myClass.getClass().getName());
        myClass.printFromLambdaFunction("Lambda Function: HelloWorld");
    }
}
```

Streams

Streams (`java.io`) VS Streams (`java.util.stream`)

Stream - Grundkonzept

- übertragen von Elementen (Daten / Objekten) zwischen einer Quelle und einem Ziel

Stream (`java.util.stream`)

- einfache (besser lesbare) Modifikation von Objekt(-Strömen)
  - Ersatz (funktionales Paradigma) für sequentielle Abarbeitung wie z.B. Schleifen
- Erweiterung des Collection Framework
- Quelle von Interface Collection, Ziel von Interface Collection, einfache Datentypen, etc (abhängig von Funktion)

Stream (`java.io`)

- übertragen von Daten zu bzw. von externen Ressourcen
- I/O --> Input / Output
- Beispiele
  - lesen aus Dateien
  - Ausgabe auf der Konsole
  - senden von Daten über das Netzwerk

Konzept

- Aufbau
  - Erzeugende-Operationen
  - Zwischen-Operationen
  - End-Operationen
- Trägheit (Laziness)
- Nicht Inteferenz

Zwischen-Operationen

- ["🍆", "🍅", "🫛"].filter("🍆") -->

Entwurfsmuster (Design Patterns)

Entwurfsmuster (Design Patterns) in der Software-Entwicklung

- Lösungsschablonen für wiederkehrende Probleme im Software-Entwurf
- zur Verbesserung der Softwarearchitektur, Strukturierung des Code und bessere Lesbarkeit
- verschiedene Arten: Erzeugungs-, Struktur- und Verhaltensmuster
- stellen keine starren Regeln, sondern flexible Richtlinien, die an die jeweilige Situation angepasst werden können

Erzeugungs-Muster (Creational Patterns)

- Erzeugung von Objekten
- entkoppeln die Konstruktion eines Objekts von seiner Repräsentation
- ermöglichen z.B. eine flexible Auswahl der konkreten Klassen zur Laufzeit
- Beispiele: factory method, Factory, Singleton, Builder

Struktur-Muster (Structural Patterns)

- erleichtern den Software Entwurf und die Strukturierung durch herstellen von Beziehungen zwischen Entitäten
- nutzen Abstraktion um komplexe, kombinierte Objekte zu erzeugen und die einzelnen Bestandteile und Strukturen flexibel zu halten
- Beispiele: Facade (Einfache Schnittstelle für komplexe Objekte), Adapter (Verbinden von inkompatiblen Schnittstellen) Kompositum, das eine Hierarchie von Objekten bildet, die als Einheit behandelt werden können

Verhaltens-Muster (Behavioral Patterns)

- erleichtern den Software Entwurf und die Strukturierung durch herstellen von Beziehungen zwischen Entitäten
- nutzen Abstraktion Sie ermöglichen eine Abstraktion, die mit anderen Lösungsansätzen kommunizieren kann2 Sie können in verschiedene Arten eingeteilt werden, wie zum Beispiel Adapter, Brücke, Dekorierer, Fassade, Fliegengewicht, Kompositum oder Proxy3 Beispiele für Strukturmuster sind: Fassade, die eine vereinfachte Schnittstelle zu einem komplexen System bietet4; Adapter, der zwei inkompatible Schnittstellen verbindet; Kompositum, das eine Hierarchie von Objekten bildet, die als Einheit behandelt werden können

Optionals

TODO... :-)

Programming Principals

DRY, KISS, ... TODO... :-)

Principal Collection

- KISS
- DRY
- FIRST