Technische Hochschule Rosenheim
Technical University of Applied Sciences

Objektorientierte Programmierung
Kapitel 3 – JUnit und Javadoc

Prof. Dr. Kai Höfig

Inhalt

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Test-Driven Development
- JUnit4
- Javadoc

Literatur: http://junit.org/junit4/

Motivation

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- *"About 15 - 50 errors per 1000 lines of delivered code." (Steve McConnell)*

Bugs, Tests, …

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Software Reliability**
  - Wahrscheinlichkeit, dass ein Software-System unter bestimmten Bedingungen keinen Fehler verursacht.
  - Messung durch *Uptime, MTTF*, …
- **Bugs**
  - Sind in komplexen (Software-)Systemen unvermeidlich.
  - Bugs können im Code versteckt sein und erst sehr viel später sichtbar werden.
- **Testen**
  - Systematischer Ansatz um Fehler aufzudecken.
  - *Failed Test*: Nachweis eines Fehlers
  - *Passed Test*: Bedeutet nur, dass kein Fehler gefunden wurde.

Testen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

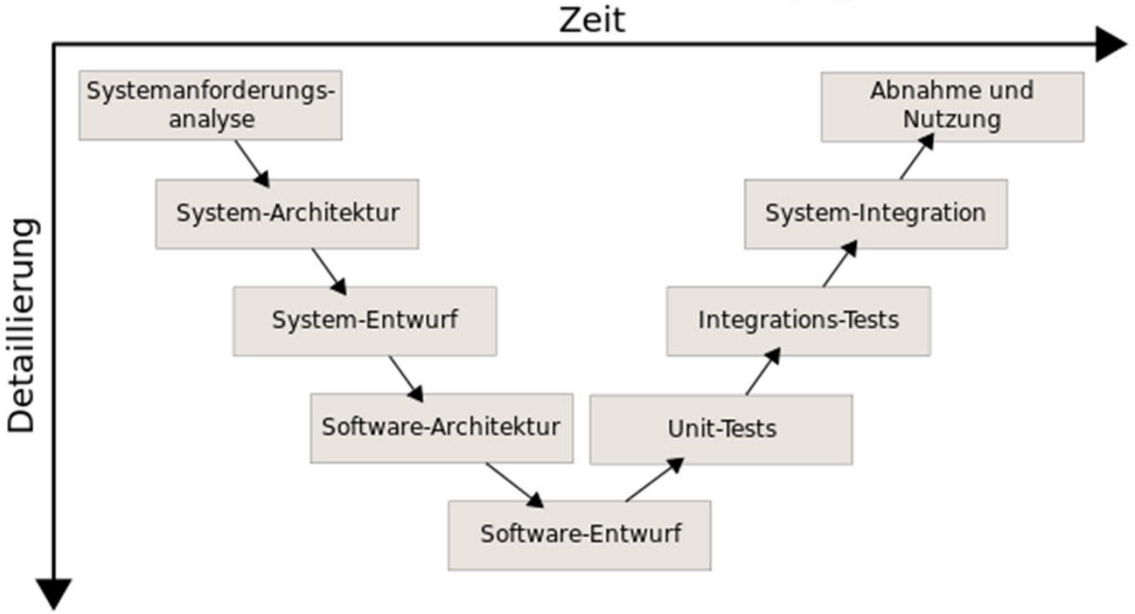
- Testen als Tätigkeit
  - kostet oft mehr Zeit als Implementieren!
  - wird oft als Aufgabe für Anfänger gesehen.

- **Grenzen** von Softwaretests
  - Unmöglich, *komplettes* System zu testen.
  - Tests können nicht beweisen, dass Software fehlerfrei ist.

- **Arten** von Tests
  - *Unit Test*: Test der Funktionalität einzelner abgrenzbarer Software-Teile.
  - *Integrationstest*: Test der Zusammenarbeit verschiedener Komponenten.
  - *Systemtest*: Test des gesamten Systems gegen die Anforderungen.
  - *Regressionstest*: Wiederholtes Ausführen von Tests nach einer Änderung.
  - *Stresstest*: Test des Systems unter großer Last.
  - …

Test-Driven Development (TDD)

- Traditionelles Vorgehen am Beispiel V-Modell
  - Testen erst ganz am Schluss!
- Nachteil bzgl. Tests
  - "Man schießt über das Ziel hinaus".
  - Tests unter Zeitdruck, da Produkt fertig werden muss.
  - Mangelnde Testbarkeit.
  - …



- **Test-Driven:** Programmierer erstellt Software-Tests konsequent **vor** der Implementierung der zu testenden Komponenten.
  - "Testen, implementieren, testen, implementieren, testen, freuen"
- Zahlreiche Vorteile:
  - Hinprogrammieren auf ein Ziel, frühes Erkennen von Problem!
  - Gute Testabdeckung, bessere SW-Qualität
  - **Programmierer kennt Schwachstellen besser als jeder andere.**

Thema dieser Vorlesung: Unit Tests

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- ***Java-Frameworks*** zum Schreiben und Ausführen automatisierter Unit Tests
  - JUnit (am weitesten verbreitet)
  - TestNG

- ***Anforderungen*** an ein Test-Framework
  - Automatisches Erzeugen von Tests nach dem Muster
    - Aufbau eines Szenarios
    - Aufruf der zu testenden Methode
    - Überprüfung ob Ergebnis korrekt ist.
  - Wiederholbar / Regression
  - Integration in IDE

- Hier: JUnit4 (Version 4)
  - Basiert auf "Annotations", siehe nächster Abschnitt!
  - In IntelliJ integriert.
  - Version 5 existiert bereits, aber noch kaum verbreitet. (Release candidate 04.02.2018)

Unit Tests mit JUnit

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Wie testet man Funktionalität einer Klasse `Foo`?
  - Erzeuge neue Klasse `FooTest`.
  - Für jede zu testende Methode: Erzeuge Methode unter Verwendung der Annotation `@Test`.
  - Verwende `assert`-Methoden um zu prüfen, ob Ergebnis der Erwartung entspricht.
    - Falls ja: Testergebnis "Pass" (grün)
    - Falls nein: Testergebnis "Fail" (rot)

Zu testende Klasse

```
public class Foo {
    public void method() {
    }
}
```

Testklasse

```
import org.junit.Test;
import static org.junit.Assert.*;

public class FooTest {
    @Test
    public void testMethod() {
        assertEquals("expected", "result");
    }
}
```

JUnit Testklasse: Muster

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
import org.junit.Test;
import static org.junit.Assert.*;

public class FooTest {
    @Test
    public void methodTest() {
        assertEquals("expected",
                     "result");
    }
}
```

Empfehlung: Falls zu testende Klasse `Foo` heißt, sollte Testklasse `FooTest` heißen

Empfehlung: Falls zu testende Methode `method` heißt, sollte Testmethode `methodTest` oder einfach `method` heißen; aussagekräftiger Name!

- Jede Methode mit Annotation `@Test` ist ein Unit Test.
- JUnit Testklassen lassen sich ähnlich wie `main`-Methode direkt starten.
- JUnit ruft automatisch jede mit @Test markierte Methode auf.
- `assert` prüft Ergebnis → grüne/rote Ampel!

JUnit Beispiel: Testen der Klasse Rational

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Teste, ob Aufruf des Default-Konstruktors Bruch $\frac{0}{1}=0$ erzeugt.

```
public class Test {
    public static void main(String[] args) {
        Rational r1 = new Rational();
        if (r1.getNumerator() != 0 || r1.getDenominator() != 1) {
            System.out.println("Error with r1");
        };
    }
}
```

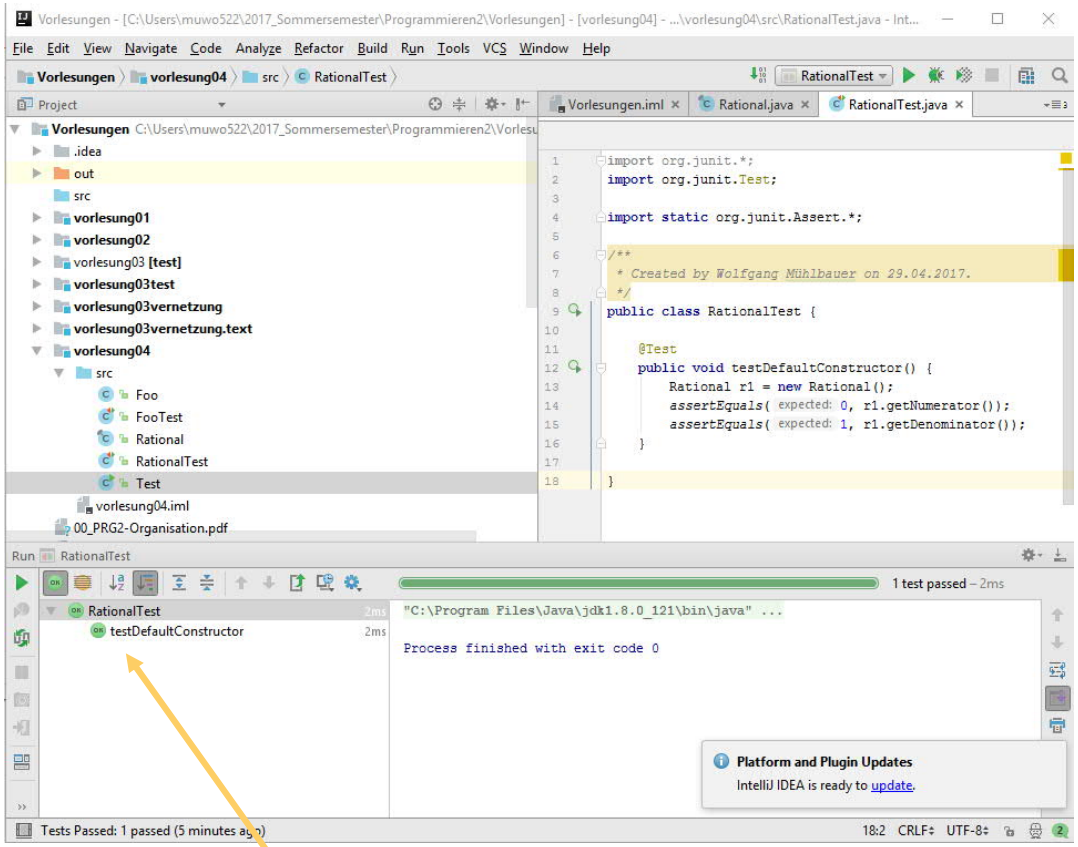
Testen über main-Methode

```
public class RationalTest {

    @Test
    public void testDefaultConstructor() {
        Rational r1 = new Rational();
        assertEquals(0, r1.getNumerator());
        assertEquals(1, r1.getDenominator());
    }

}
```

Testen über JUnit

Testen mit JUnit4 und IntelliJ

Technische Hochschule Rosenheim
Technical University of Applied Sciences



Alle Tests erfolgreich!

Installation:

1. File → Projekt Structure
2. Libraries
3. +
4. From Maven
5. Junit 4.12

Testfallerzeugung:
Framework durchsacht Testklasse nach Annotation `@Test`.

Testlauf:
Gesammelte Testfälle werden voneinander unabhängig durchgeführt.

JUnit und IntelliJ: Tipps

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Automatisches Generieren der Testklasse**
  - Cursor auf Klassendefinition und im Kontextmenü: "*Go To … Test*"
  - Cursor auf Klassendefinition in Menü "*Navigate … Test*"

- **Erzeugen der Testmethode, z.B.**
  - Manuell
  - Oder z.B. Cursor in Methodendeklaration setzen, dann "`Alt+Enter` → *Generate Missed Test Methods"*

- **Hinzufügen von fehlenden Import Statements**
  - `Alt+Enter`

- **Tests laufen lassen**
  - Im Projektfenster auf die Testklasse klicken, dann im Kontextmenü (rechte Maustaste) "*Run*" auswählen.
  - Bei "Fail": IntelliJ zeigt erwarteten Wert und "gemessenen" Wert an.

JUnit: `Assert`-Methoden

Technische Hochschule Rosenheim
Technical University of Applied Sciences

`assertTrue(test)`	fails if the boolean test is `false`
`assertFalse(test)`	fails if the boolean test is `true`
`assertEquals(expected, actual)`	fails if the values are not equal
`assertSame(expected, actual)`	fails if the values are not the same (by ==)
`assertNotSame(expected, actual)`	fails if the values are the same (by ==)
`assertNull(value)`	fails if the given value is not `null`
`assertNotNull(value)`	fails if the given value is `null`
`fail()`	causes current test to immediately fail

- Jeder Methode kann man auch einen String übergeben, der im Fehlerfall angezeigt wird
  - Z.B.: `assertEquals("message", expected, actual)`
  - Wichtig: Message steht immer am Anfang.

Übung: Was ist hier ungünstig/falsch?

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```java
public class RationalTest {

    @Test
    public void test5() {
        Rational r2 = new Rational(1, 2);
        assertEquals(r2.getNumerator(), 1);
        assertEquals(r2.getDenominator(), 2);
    }
}
```

Der erwartete Wert sollte immer links stehen! (sonst Missverständnisse bei der Anzeige in IntelliJ)

- **Verbesserung 1:** Hinzufügen von Nachrichten um Fehler bei Auftreten einfacher zu identifizieren
  - Bsp.: `assertEquals("Numerator value", 1, r2.getNumerator())`
- **Verbesserung 2:** Aussagekräftige Testnamen
  - `testDefaultConstructor(…)` anstatt `test5(…)`
- **Hinweis:** Bei Verwendung von `assertEquals()` **muss** eine für die Anwendung "passende" `equals()` Implementierung vorhanden sein.
  - Siehe übernächstes Kapitel!

Testen mit Timeouts

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
@Test(timeout = 5000)
public void name() {...}
```

- Die obige Methode ergibt "FAIL", falls der Testcase nicht innerhalb von 5000 ms beendet wird.

```
private static final int TIMEOUT = 2000;
...
@Test(timeout = TIMEOUT)
public void name() {...}
```

- Bei obigem Code "FAIL", falls nach 2000 ms nicht beendet.

- ***Hinweis***: Falls eine zu testende Methoden endlos läuft, endet auch der Testcase nicht. Alle weiteren noch nicht ausgeführten Tests werden dann gar nicht gestartet.

Ausblick: Testen von Ausnahmen / Exceptions

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
@Test(expected = ExceptionType.class)
public void name() {
    ...
}
```

- Ergibt "Pass" falls die Ausnahme / Exception tatsächlich eintritt.
- Sollte verwendet werden, um zu testen, ob bestimmte Fehler auch wie erwartet eintreten.
- Details: siehe Kapitel über "Exceptions"

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
    int[] array = new int[4];
    int i = array[4];   // should fail
}
```

Setup und Teardown

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Tests sollten voneinander unabhängig sein.
- Jeder Test sorgt dafür, dass Initialzustand hergestellt ist.
- Um Quellcodeduplizierung zu vermeiden, gibt es Spezialmethoden.

- Methode, die ***vor/nach*** Ausführung ***jedes*** Testcases aufgerufen wird.

```
@Before
public void setUp() { ... }
@After
public void tearDown() { ... }
```

- Methode, die ***nur einmal zu Beginn*** aufgerufen wird und ***nur einmal nachdem ALLE*** Test Cases beendet sind, aufgerufen wird.
  - Achtung: Statische Methode!

```
@BeforeClass
public static void beforeClass() { ... }
@AfterClass
public static void afterClass() { ... }
```

Setup und Teardown: Übung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Die folgenden JUnit Tests werden ausgeführt.
- Wie lautet die Ausgabe auf der Konsole?

```
public class FixtureDemoTest
{   @BeforeClass public static void beforeClass()  {
        System.out.println( "@BeforeClass" );
    }
    @AfterClass public static void afterClass()    {
        System.out.println( "@AfterClass" );
    }
    @Before public void setUp()     {
        System.out.println( "@Before" );
    }
    @After public void tearDown()   {
        System.out.println( "@After" );
    }
    @Test public void test1()    {
        System.out.println( "test 1" );
    }
    @Test public void test2()    {
        System.out.println( "test 2" );
    }
}
```

Allgemeine Richtlinien

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Eingrenzung der zu testenden Eingaben, Parameter, etc.
  - Randfälle: Positiv, null, negative Zahlen
  - Linkes und rechtes Ende eines Arrays
  - "Leerfälle": 0, -1, null, leeres Array

- Teste das Verhalten in Kombinationen
  - `add()` funktioniert normal, aber nicht wenn zuvor `remove()` aufgerufen wurde.
  - Möglicherweise schlägt erst der 2. Aufruf einer Funktion fehl.

- Teste soweit möglich nur 1 Sache gleichzeitig.

- Tests sollten soweit als möglich Logik vermeiden.
  - Kein if/else, Schleifen, etc. im Code der Testmethode.

- Tests sollten voneinander unabhängig sein.
  - Es sollte keinen Unterschied machen ob Test A vor Test B ausgeführt wird.

Allgemeine Richtlinien

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Eingrenzung der zu testenden Eingaben, Parameter, etc.
  - Randfälle: Positiv, null, negative Zahlen
  - Linkes und rechtes Ende eines Arrays
  - "Leerfälle": 0, -1, null, leeres Array

- Teste das Verhalten in Kombinationen
  - `add()` funktioniert normal, aber nicht wenn zuvor `remove()` aufgerufen wurde.
  - Möglicherweise schlägt erst der 2. Aufruf einer Funktion fehl.

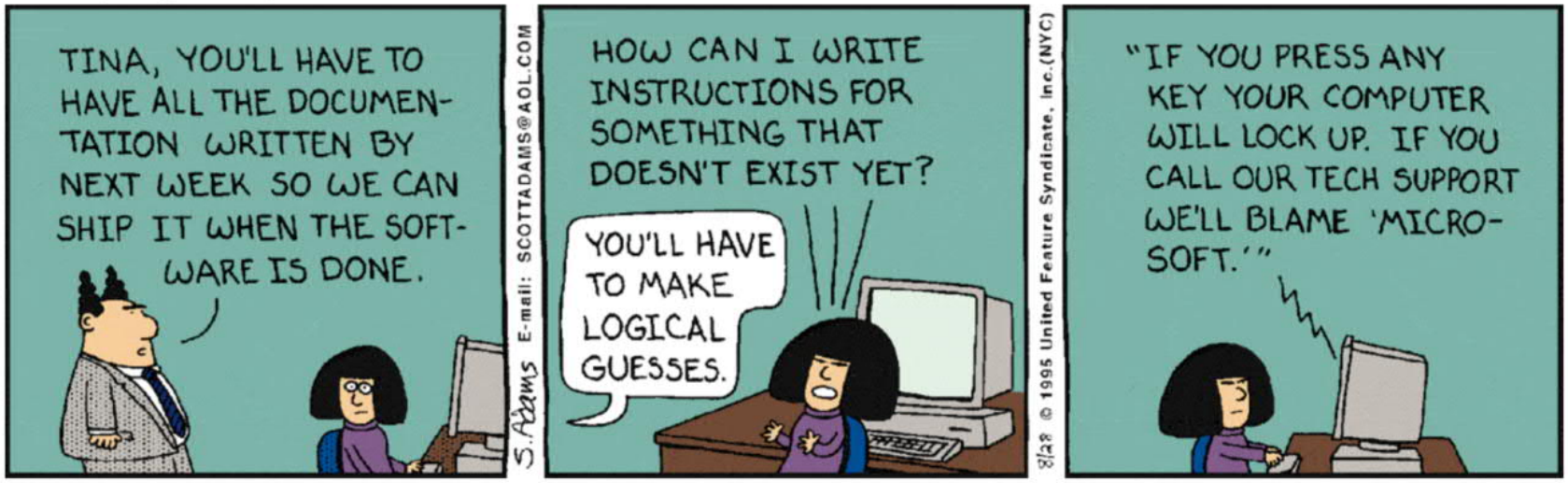
- Teste soweit möglich nur 1 Sache gleichzeitig.

- Tests sollten soweit als möglich Logik vermeiden.
  - Kein if/else, Schleifen, etc. im Code der Testmethode.

- Tests sollten voneinander unabhängig sein.
  - Es sollte keinen Unterschied machen ob Test A vor Test B ausgeführt wird.

Zur Erholung …

Technische Hochschule Rosenheim
Technical University of Applied Sciences



Dokumentation mit Javadoc

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Motivation**
  - Dokumentation wird bei Codeänderungen oft nicht aktualisiert.
  - Dokumentation wird unter Zeitdruck oft vernachlässigt

- **Lösung**
  - Integration von Quelltext und Dokumentation, d.h. Quelltext und Dokumentation in **gleicher** Datei
  - Erweiterung des Konzepts der Blockkommentare

- **Dokumentationsgenerator:** `javadoc`
  - Erzeugt zu jeder `.java` Datei eine `.html` Datei, mit Beschreibung von Klasse, Interface, Methoden, etc.
  - Dokumentation mittels spezieller Kommentare
    - Stehen im Quelltext unmittelbar vor dem zu Dokumentierenden
    - Beginnen mit `/**` und enden mit `*/`
    - Können aus mehreren Zeilen bestehen; erster Satz (bis zum ersten Punkt) ist Kurzbeschreibung

Javadoc Beispiel

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```java
/**
 * Einfache Implementierung für rationale Zahlen.
 *
 * Rationale Zahlen werden über Zähler und Nenner dargestellt.
 *
 * @author Professoren der Informatik
 * @version 1.1
 */
public class Rational {
    private long numerator;  private long denominator;

    /**
     * Rationalzahl mit Zähler und Nenner vom Typ long
     *
     * @param num Zähler
     * @param den Nenner
     */
    public Rational(long num, long den) {}
    }

    /**
     * Addiert zwei rationale Zahlen.
     *
     * @param val rationale Zahl, die zu dieser addiert werden soll.
     * @return Eine neue Rationalzahl als Ergebnis der Operation
     */
    public Rational add(Rational val) {
        return null;        // just to shorten the code
    }
}
```

Javadoc für Klassendeklaration

Javadoc für Konstruktor

Javadoc für Methode

Aufbau eines Javadoc-Kommentars

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Dokumentation von
  - Klassen und Interfaces
  - Methoden
  - Attribute (Datenelemente)

- Inhalte von Javadoc-Kommentaren
  - Beschreibung (Zusammenfassung und Details)
  - Tags → Markieren von Schlüsselinformationen

- Tags
  - Aufbau: *@keyword [parameter] text*
    - *keyword* bezeichnet Schlüsselinformation
    - *text* steht für Fließtext

  - Unterschiedliche Tags für
    - Klassen und Interfaces
    - Methoden

  - Keine Tags für Datenelemente

  - Javadoc-Tags sind keine Annotationen

Javadoc: Die wichtigsten Tags

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Tags für **Klassen und Interfaces**
  - *@author text*
    - Name des Autors bzw. Autoren
  - *@version text*
    - Version des Quelltextes
- Tags für **Methoden**
  - *@param name text*
    - Bedeutung des Parameters *name*
    - Wiederholung für jeden Parameter
  - *@return text*
    - Bedeutung des Ergebnisses der Methode
    - Fehlt bei `void`-Methoden und Konstruktoren
  - *@throws exceptionclass text*
    - Hinweis auf evtl. geworfene Ausnahmeklasse (siehe später)
    - Für jede Ausnahme (Exception) wiederholt

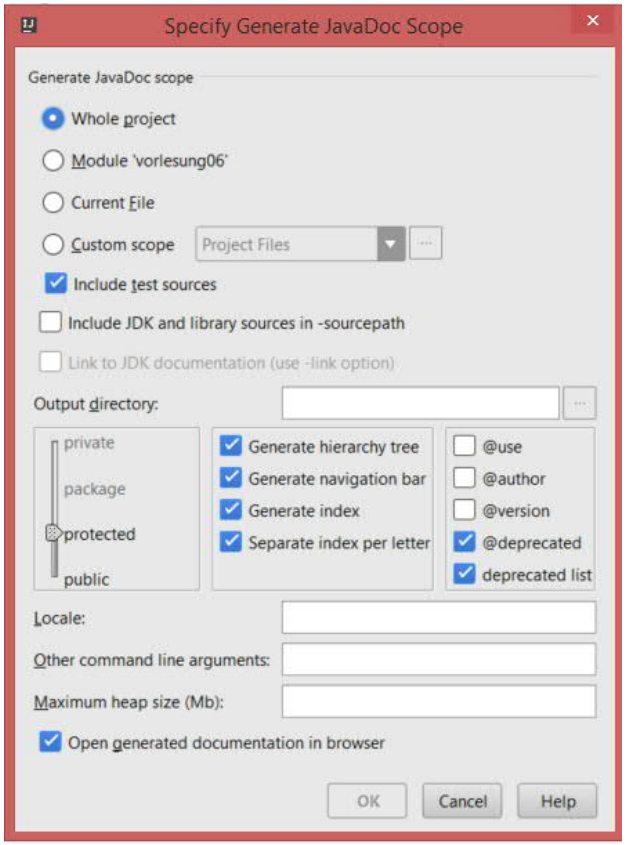
Erzeugen der Javadoc-Dokumentation

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Spezieller Compiler `javadoc` ist Bestandteil des JDK
  - Aufruf über Kommandozeile möglich.

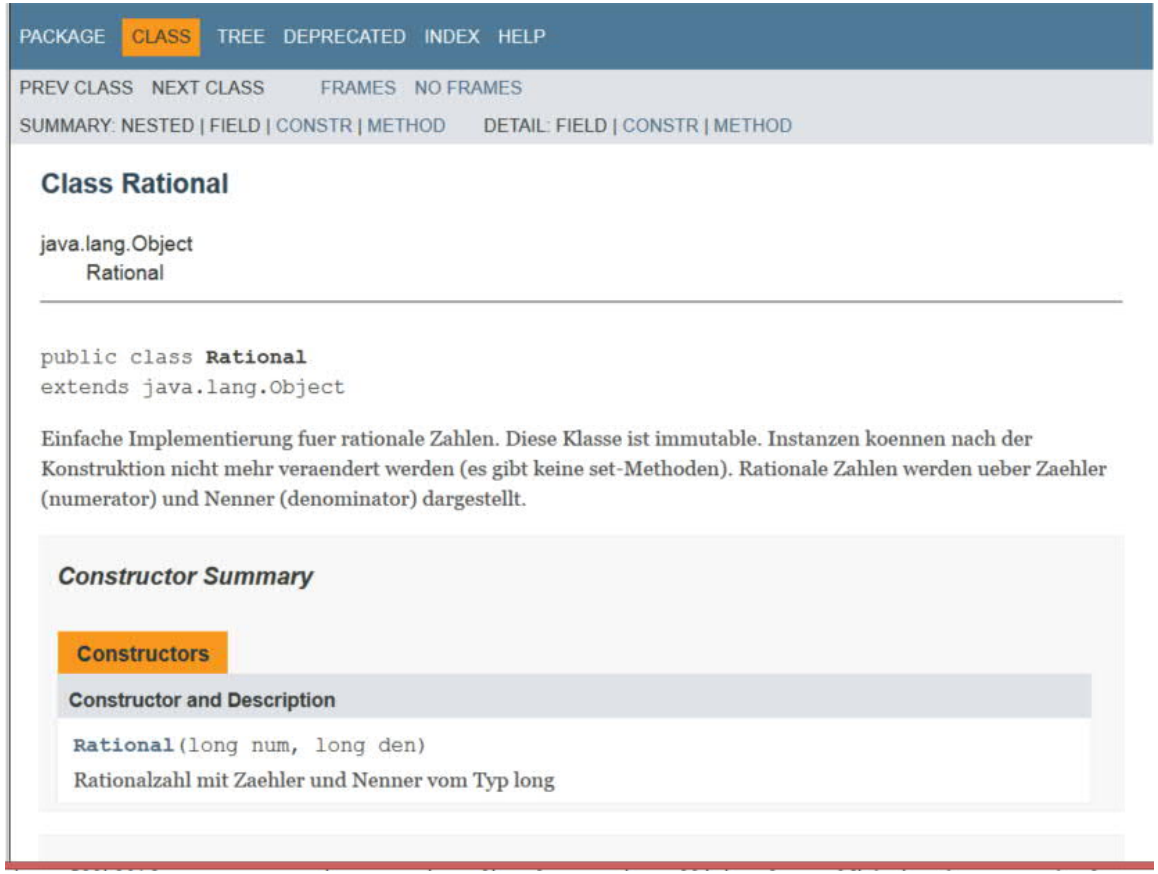
- Ergebnis mit jedem Webbrowser lesbar
  - Pro Klasse eine HTML-Seite

- IntelliJ
  - *Tools → Generate Javadoc*



Erzeuge Javadoc Dokumentation

Technische Hochschule Rosenheim
Technical University of Applied Sciences



PACKAGE CLASS TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class Rational

java.lang.Object
Rational

```
public class Rational
extends java.lang.Object
```

Einfache Implementierung fuer rationale Zahlen. Diese Klasse ist immutable. Instanzen koennen nach der Konstruktion nicht mehr veraendert werden (es gibt keine set-Methoden). Rationale Zahlen werden ueber Zaehler (numerator) und Nenner (denominator) dargestellt.

Constructor Summary

Constructors

Constructor and Description
`Rational(long num, long den)` Rationalzahl mit Zaehler und Nenner vom Typ long

Zusammenfassung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Test-Driven Development
  - Erst Tests schreiben, dann implementieren!
  - Verbessert Qualität der Software
- Annotationen
  - Hinterlegen von Metainformationen im Programmcode
- JUnit4
  - Bibliothek zum einfachen Erstellen und Ausführen von JUnit Tests unter Java.
- Javadoc
  - Dokumentation eines Programmes innerhalb des Codes.