Technische Hochschule Rosenheim
Technical University of Applied Sciences

Programming Basics – WiSe21/20

Unit Tests with Junit, Documentation with Javadoc

Prof. Dr Silke Lechner-Greite

Table of contents – planned topics

Technische Hochschule Rosenheim

Technische Hochschule Rosenheim

Chapter 10: Unit Tests with JUnit, Documentation with Javadoc

Motivation

- *"About 15 - 50 errors per 1000 lines of delivered code."*

  (Steve McConnell)

Bugs, Tests, …

Technische Hochschule Rosenheim

- **Software Reliability**
  - The probability that a software system will not cause an error under specific conditions.
  - Measurement by *uptime, Mean Time To Failure*, …
- **Bugs**
  - Are unavoidable in complex (software) systems.
  - Bugs can be hidden in the code and only become visible much later.
- **Tests**
  - Systematic approach to detecting errors.
  - *Failed Test*: proof of an error
  - *Passed Test*: only means that no error was found.

Tests

Technische Hochschule Rosenheim

Testing as an activity

- Often takes more time than implementation!
- Is often seen as a task for beginners.

Limitations of software testing

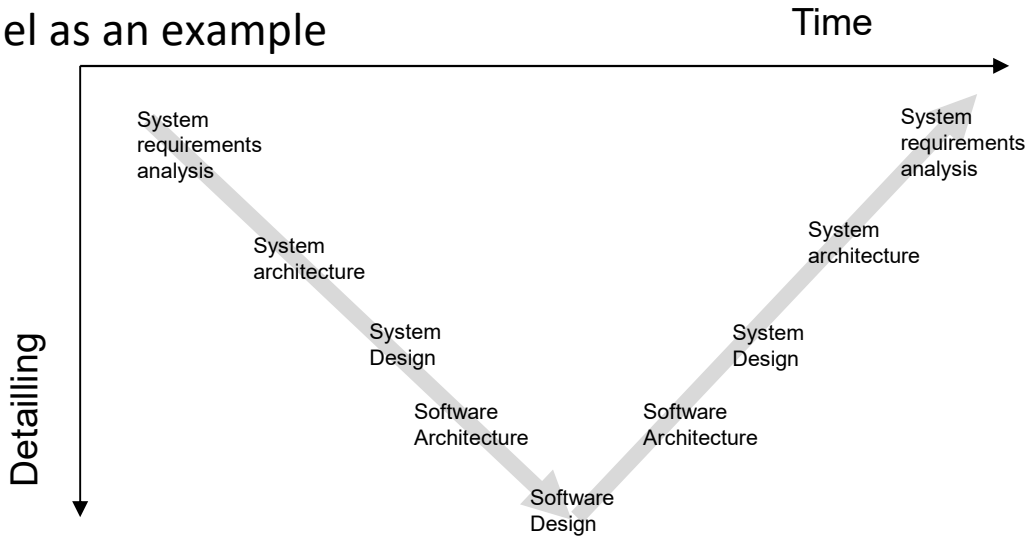
- Impossible to test a *complete* system.
- Tests cannot prove that software is error-free.

Types of test

- *Unit test*: tests the functionality of individual delimitable software components.
- *Integration test*: tests the cooperation between different components.
- *System test*: tests the entire system against the requirements.
- *Regression test*: re-run of tests after a change.
- *Stress test*: tests the system under a heavy load.
- …

Test-driven development (TDD)

- Traditional approach using the V-model as an example
  - Testing only right at the end!
- Disadvantage of tests
  - "You take things too far."
  - Tests under time pressure, as product needs to be finished.
  - Lack of testability.
  - …



- **Test-driven:** programmers consistently create
- software tests **before** implementing the components to be tested.
  - "Test, implement, test, implement, test, be happy"
- Numerous advantages:
  - Programming towards a goal, early detection of problems!
  - Good testing coverage, better software quality
  - **Programmers know weak spots better than anyone else.**

Topic of this lecture: unit tests

Technische Hochschule Rosenheim

- ***Java frameworks*** for writing and executing automated unit tests
  - JUnit (most widely used)
  - TestNG
- ***Requirements*** for a test framework
  - Automatically generate tests according to the pattern
    - Build a scenario
    - Call the method to be tested
    - Check whether the result is correct.
  - Repeatable / regression
  - Integration in IDE
- Here: JUnit 5 (Version 5)
  - Based on annotations and assertions
  - Integrated in IntelliJ.
  - The new JUnit 5 framework can process previous versions such as JUnit 3 and JUnit 4.

Goal of unit tests?

Technische Hochschule Rosenheim

- Already write tests for each method during implementation
- At the time of programming the method, you know best exactly what should be implemented, and what you have implemented.
- Simplify the work:
  - Reduce debugging times
  - Eliminate the assumption: “the method works”. The test proves that it works, or at least that no error was found.
  - Already exclude errors in basic methods to reduce follow-up errors due to combining using methods
  - Build trust: you know how the code behaves
  - Create understanding: compare results with expectations

What are the tasks of unit tests?

Technische Hochschule Rosenheim

- Providing evidence:
  - Show WHAT should be achieved by the **test**
  - Show WHAT the **functionality** of the source code is
  - Show that the code does what you wanted it to

- Know borderline cases: what if it actually doesn't work like that? (e.g. no hard disk space, network connection was lost, exceptions occur, etc.)
  - Make sure the code FUNCTIONS and WORKS CORRECTLY
  - Code reliability: I know the strengths, I know the weaknesses / limitations
  - This keeps limitations transparent
  - Makes teamwork easier

- Unit tests as documentation:
  - The intended use of the code is clarified
  - Testing the borderline cases / boundary conditions shows what can be expected of the code

Technische Hochschule Rosenheim

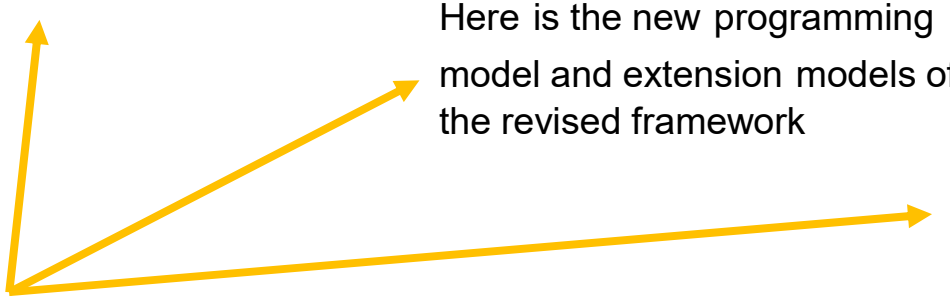
Chapter 10: Unit Tests with JUnit, Documentation with Javadoc

JUnit 5 architecture

Technische Hochschule Rosenheim

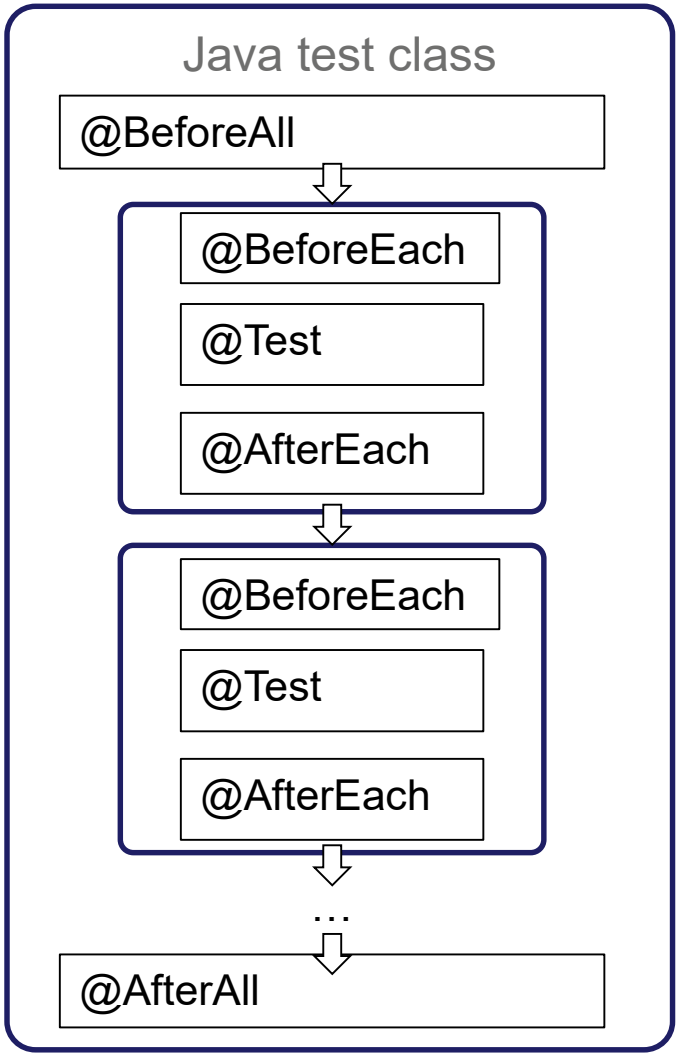
JUnit 5

Vintage	Jupiter	Platform
Test engine, so that JUnit 3 and JUnit 4 tests can be run	Test engine, so that JUnit 5 tests can be run	Initiates test frameworks on the JVM.
	Here is the new programming model and extension models of the revised framework	Test engine API for the development of a test framework, e.g. for external third-party test framework providers.
		Console launcher – can be started from the command line
		JUnit 4 based runner
		Support for IntelliJ, Eclipse, NetBeans, Visual Studio Code.



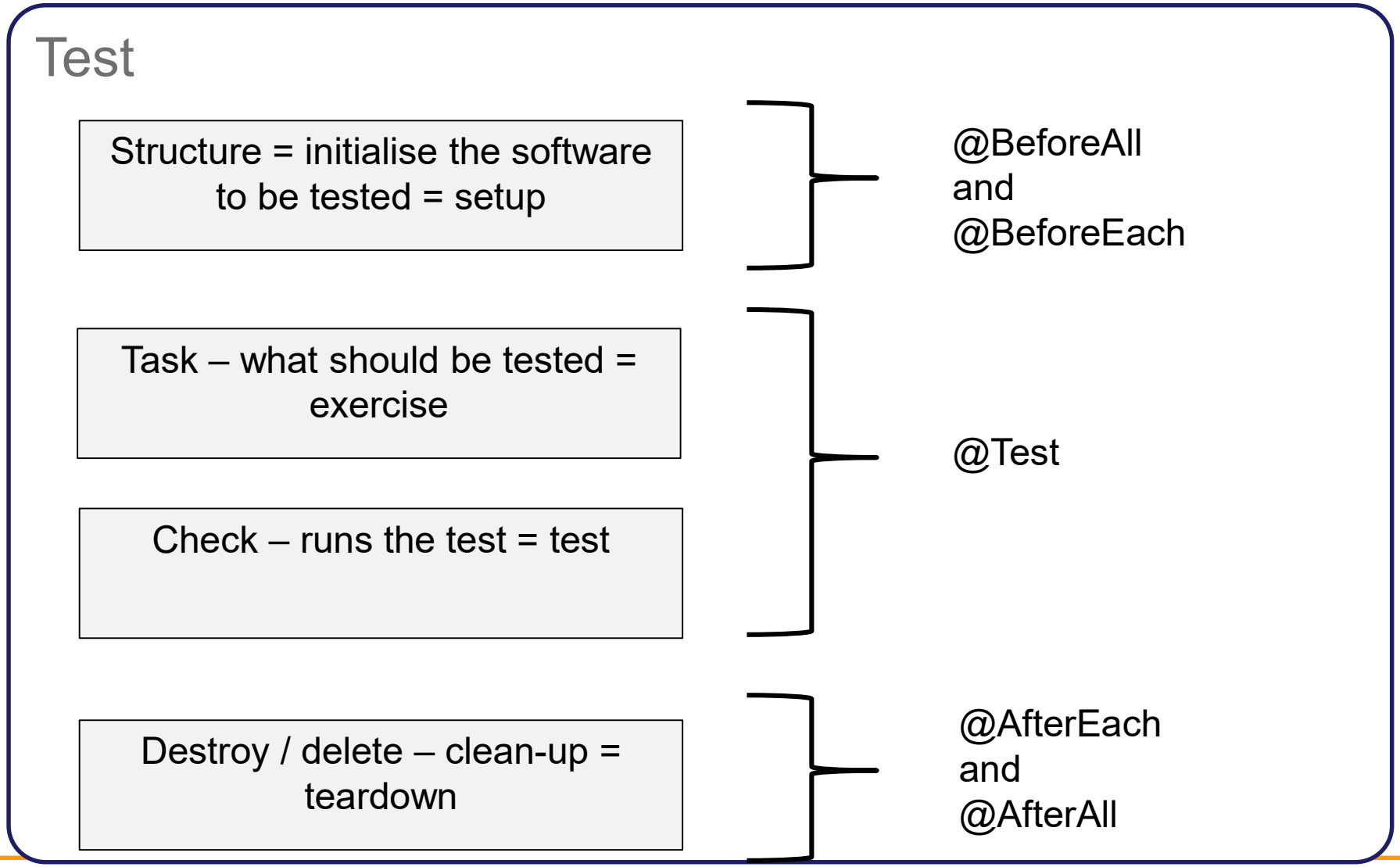
- JUnit 5 consists of several subprojects:
  JUnit Platform + JUnit Jupiter + JUnit Vintage

JUnit 5 Standard Test



- JUnit 5 also uses annotations to identify various methods within the test life cycle
- The Jupiter package (org.junit.jupiter.api.*) contains all annotations
- @BeforeEach: run before each test
- @AfterEach: run after each test
- @BeforeAll: run before all test methods of the current class
- @AfterAll: run after all test methods of the current class

Structure of a test



Unit Tests with JUnit 5

Technische Hochschule Rosenheim

- How do we test the functionality of a class **Foo**?
  - Create a new class **FooTest**.
  - For each method to be tested: create a method using the annotation **@Test.**
  - Use **assert** methods to check whether the result matches the expectation.
    - If yes: test result "pass" (green)
    - If no: test result "fail" (red)

Class to be tested

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

Test class

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

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

JUnit 5 test class: template

Technische Hochschule Rosenheim

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


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

Recommendation:
If the class to be tested is called `Foo`, the test class should be named `FooTest`

Recommendation:
If the method to be tested is called `method`, the test method should be `methodTest` or simply `method`; use a meaningful name!

- Every method with the annotation `@Test` is a unit test.
- JUnit test classes can be started directly, similar to the `main` method.
- JUnit automatically calls every method marked with @Test.
- `assert` checks the result → green/red light!

JUnit 5 example: test the Rational class

Technische Hochschule Rosenheim

- Test whether call of default constructor fraction generates $\frac{0}{1}$=0.

```
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");
          };
     }
}
```

Testing using main method

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

class RationalTest {


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

Testing using JUnit 5

Some facts about JUnit 5

Technische Hochschule Rosenheim

- Import statements start with **org.junit.jupiter**….
- e.g. **import static org.junit.jupiter.api.Assertions.***
- Assertions are static methods. Import static allows them to be called directly without having to type Assertions.assertEqual or the like.

- Visibility: in JUnit 5, a test class and a method within it **no longer have to be made public** → package visibility is sufficient.

- A method which is to be run as a test is annotated with **@Test**.

- There are help areas that set up a test (**setup**) and then destroy it again (**teardown**).
  - @BeforeAll - runs once; runs before the tests marked @BeforeEach
  - @BeforeEach - is performed before each test
  - @AfterEach - is performed after each test
  - @AfterAll - runs once; runs after all the tests marked @AfterEach

Some facts about JUnit 5 (cont.)

Technische Hochschule Rosenheim

- A new instance is created for each test. There is no obvious assignment of which instance @BeforeAll / @AfterAll methods can be called for, so they must be static. → Tests cannot share their state through non-static fields in the test class.

- With JUnit 5, you can switch to a single instance for all test methods:

```
import static org.junit.jupiter.api.TestInstance.*;
@TestInstance(Lifecycle.PER_CLASS)
public class TestLifecycle{ … }
```

- Tests can be “disabled” (also in conjunction with conditions):

```
@Test
@Disabled
void disabledTest() {
   assertTrue(false);
}
```

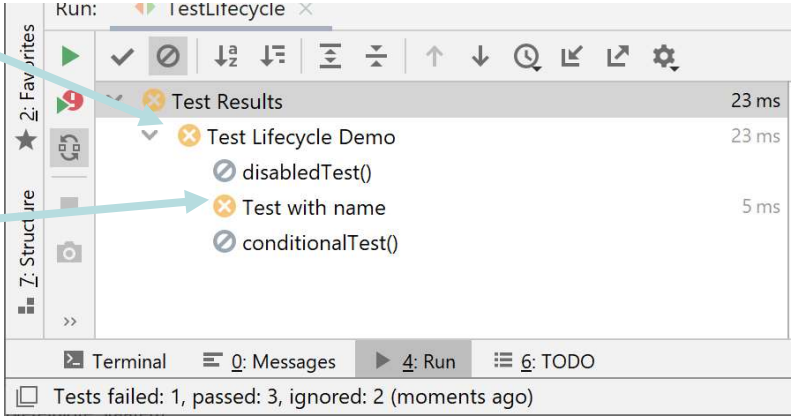
```
@Test
@DisabledOnOs(OS.WINDOWS)
@DisabledOnJre(JRE.JAVA_8)
void conditionalDisabledTest() {
   assertTrue(false);
}
```

Some facts about JUnit 5 (cont.)

Technische Hochschule Rosenheim

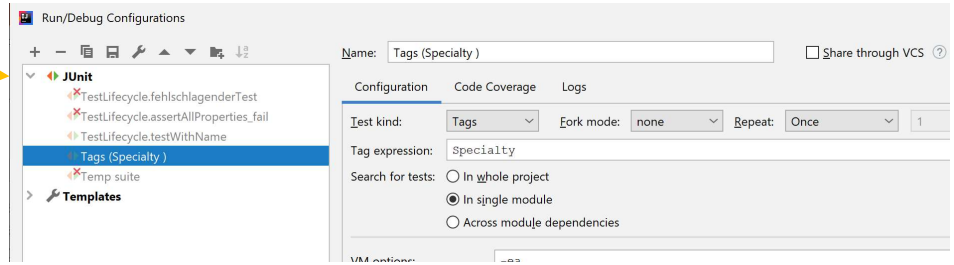
- Test classes and individual tests can be given names:

```
@DisplayName("Test Lifecycle Demo")
public class TestLifecycle {
   // ...
   @Test
     @DisplayName("Test with name")
   void testWithName() { }
```



- Test classes can be assigned a “tag” (tagging): in this way, the IDE knows that only tests with a certain “tag” are executed. This is useful if you want to group tests, e.g. user interface, database, etc. (tags must be specified in the IDE, IntelliJ: under Configuration tab

```
@Tag("Specialty")
public class TestLifecycle {
   // …
```



- If the test class does not have this tag, then it cannot be run.

Annotations –JUnit 5

Technische Hochschule Rosenheim

Function	JUnit 5
Annotation package	org.junit.Jupiter.api
Declaration of a test	@Test
Setup for all tests	@BeforeAll
Setup for one test	@BeforeEach
Teardown for one test	@AfterEach
Teardown for all tests	@AfterAll
Deactivate a test method or class	@Disable
Nested tests	@Nested
Repeated tests	@Repeated
Runners / rules	Replaced by @ExtendWith

A comprehensive list of JUnit 5 annotations can be found here:
https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations

Assertions

Technische Hochschule Rosenheim

- Assertion - logical proposition (predication)
- Assertions ensure that the desired properties of the method to be tested actually occur, and if not, the test is shown as "fail".
- JUnit 5 offers classic (as before) and extended (i.e., new) assertions:
  - Classic: the main purpose here is to compare properties, e.g., whether two instances match (expected vs. actual) or whether one instance is not equal to null.

```
@Test
void assertWithComparison() {
   Rational expected = new Rational(3,4);
   Rational actual = expected;
   assertEquals(expected, actual);
   assertEquals(expected, actual, "Should be the same .");
   assertNotSame(expected, actual, "Obviously not the same instance.");
}
```

JUnit: “classic” `assert` methods

Technische Hochschule Rosenheim

`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

- A string can be passed to each method, which is displayed in the event of an error
  - e.g.: `assertEquals(`**expected**`, `**actual**`, "message")`
  - Important: **order is** expected, actual – not the other way around, otherwise there will be confusion during the test run!
  - The "messages" are put at the end!

Assertions (cont.)

Technische Hochschule Rosenheim

- Extended (only a small extract):
  - A test may fail from the outset with an error message:

```
@Test
void failedTest() {
  fail("Failed, whatever happens.");
}
```

  - assertAll: executes a variable number of assertions before it returns a possible error:

```
@Test
void assertAllProperties () {
    Rational r = new Rational(7, 8);
  System.out.println(r.toString());
  assertAll("Rational",
            () -> assertEquals(7, r.getNumerator()),
            () -> assertEquals(8, r.getDenominator()),
            () -> assertEquals("Fraction",r.toString())
    );
}
```

Good for testing several related properties

  - Assertions now accept lambda expressions () -> { do sth }

More assertions: https://junit.org/junit5/docs/current/api/

Good to know

Technische Hochschule Rosenheim

- The concepts presented form the fundamental basis of unit tests with JUnit 5.

- There are numerous other possibilities, such as
  - The testing of interfaces … a test interface is created for each interface implemented. Each test class that tests an implemented class implements the test interface and has access to all test methods of the interface.

  - Nested tests – test classes can be nested, so that multiple test classes are executed - @Nested.

  - Parameterised tests – to avoid duplicate test logic. These tests can be run multiple times with different arguments; this is achieved by including junit-jupiter-params and using the annotation @ParameterizedTest.

  - Dynamic tests – tests are declared at runtime - @TestFactory

Parameterised test - example

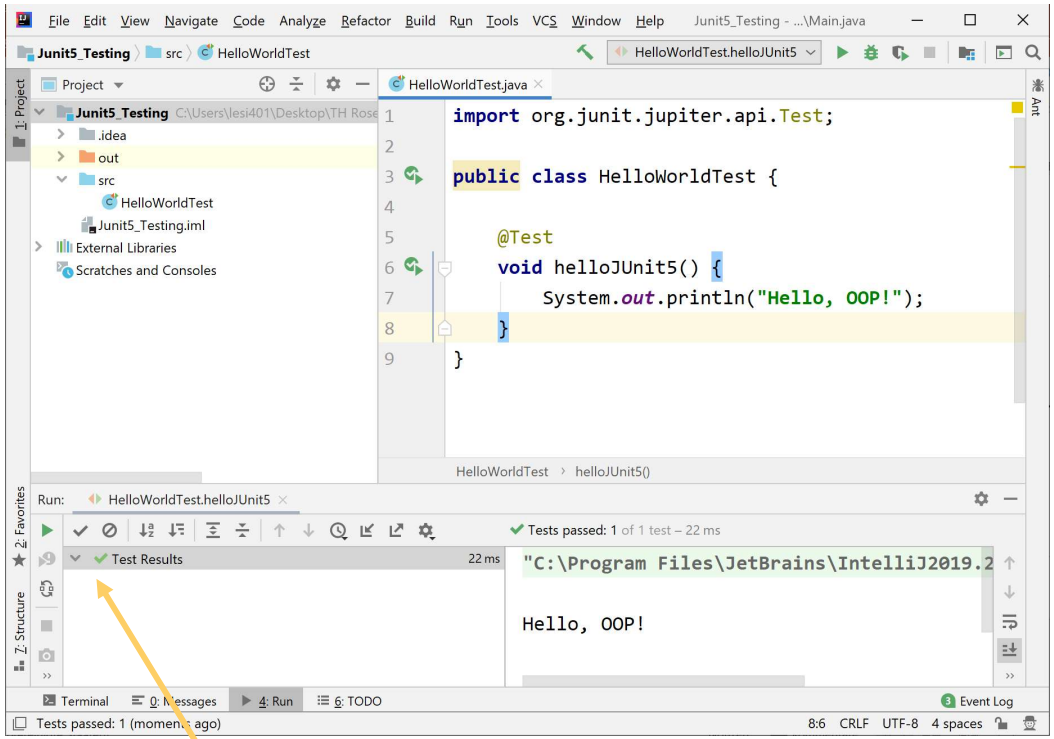
Technische Hochschule Rosenheim

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

import static org.junit.jupiter.api.Assertions.assertTrue;

public class parameterizedTest {
  @ParameterizedTest
    @ValueSource(strings = {
        "otto",
        "Do geese see God?",
        "Risotto, Sir?",
        "Radar",
        "Some rubbish",
        "Racecar"
  })
  void palindromes(String palims) {
    assertTrue(Palindrom.istPalindrom4(Palindrom.filter(palims)));
    }
}
```

Testing with JUnit 5 and IntelliJ



All tests successful!

Installation (for each project):

1. File → Project Structure
2. Libraries
3. +
4. From Maven
5. org.junit.jupiter:junit-jupiter:**5.5.2**
6. Apply - OK

Test case creation:
Framework searches test class for annotation `@Test`.

Take the newest version!

JUnit and IntelliJ: tips

Technische Hochschule Rosenheim

- **Automatically generate the test class**
  - Cursor on class definition and in the context menu: "*Go To … Test*"
  - Cursor on class definition in menu *"Navigate … Test*"
- **Create the test method, e.g.**
  - Manually
  - Or, for example, place the cursor in the method declaration, then "`Alt+Enter` → *Generate Missed Test Methods"*
- **Add missing import statements**
  - `Alt+Enter`
- **Run tests**
  - In the project window, click on the test class, then select "*Run*" from the context menu (right mouse button).
  - In the event of "Fail": IntelliJ shows the expected value and "measured" value.

Exercise: what's wrong here?

Technische Hochschule Rosenheim

```
public class RationalTest {

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

Exercise: what's wrong here?

Technische Hochschule Rosenheim

```
class RationalTest {

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

The expected value should always be on the left!
(otherwise misunderstandings when displaying in IntelliJ)

Public does not have to be used.

- **Improvement 1:** Assign a display name
  - `@DisplayName("Test with name")`
- **Improvement 2:** Meaningful test names
  - `testDefaultConstructor(…)` instead of `test5(…)`
- **Improvement 3:** Add messages to help identify errors more easily when they occur
  - Example.: `assertEquals(1, r2.getNumerator(), "Numerator value")`

Coding

Technische Hochschule Rosenheim

- Testing static and object-oriented methods
- Code generation
- Method with sum
- Method with boolean
- Method with out
- Own object variables in the class
- @Before for initialisation

Live exercise

Technische Hochschule Rosenheim

```
public class Fraction {
   private int z,n;
   public Fraction(int z, int n){
      this.z=z;
      this.n=n;
   }
   public Fraction multiply(Fraction b){
      return new Fraction(z*b.getZ(),n*b.getN());
   }
   public double getDouble(){
      return (double)z/n;
   }
   public int getZ() { return z;}
   public int getN() { return n;}
}
```

Create a JUnit test to test the method `multiply()`

Testing with timeouts – JUnit 5!

Technische Hochschule Rosenheim

```
@Test
void timeoutNotExceeded() {
  assertTimeout(ofMinutes(2), () -> {
    // Do something that needs less than 2 minutes.
  });
}
```

- The method above returns "FAIL" if the test case is not terminated within 2 minutes. The method below returns "FAIL" because the test case simulates a scenario > 10 ms.

- ***Note***: If a method to be tested runs indefinitely, the test case does not end either. All other tests that have not yet been run will then not be started at all.

```
@Test
void timeoutExceeded() {
  assertTimeout(ofMillis(10), () -> {
    Thread.sleep(100);
   });
}
```

Outlook: testing exceptions – JUnit 5

Technische Hochschule Rosenheim

```
@Test
void divideByzero() {
    Rational r1 = new Rational(3,4);
    Rational r2 = new Rational(0,0);
    Exception exception = assertThrows(ArithmeticException.class, () ->
       r1.divide(r2));
  assertEquals("/ by zero", exception.getMessage());
}
```

- Returns "Pass" if the exception actually occurs.
- Should be used to test whether certain errors also occur as expected.
- Details: see chapter on "Exceptions"

```
@Test
public void testBadIndex() {
  int[] array = new int[4];
  int index = 4;
    Exception exception = assertThrows(ArrayIndexOutOfBoundsException.class,
            () -> {int i = array[4];});
  assertEquals("Index out of bounds" + index, exception.getMessage());
}
```

Setup and teardown

Technische Hochschule Rosenheim

- Tests should be independent of each other.
- Each test ensures that the initial state is established.
- To avoid source code duplication, there are special methods.

- Method that is called ***before/after each*** test case is run.

```
@BeforeEach
void setUp() { ... }
@AfterEach
void tearDown() { ... }
```

- Method that is ***only called once at the beginning*** and ***only called once after ALL*** test cases have ended.
  - Note: static method!

```
@BeforeAll
static void beforeClass() { ... }
@AfterAll
static void afterClass() { ... }
```

Setup and teardown: exercise

Technische Hochschule Rosenheim

- The following JUnit tests are run.
- What is the output to the console?

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

General guidelines

Technische Hochschule Rosenheim

- Limitation of the inputs, parameters, etc. to be tested.
  - Boundary value cases: positive, null, negative numbers
  - Left and right end of an array
  - "Empty cases": 0, -1, null, empty array
- Test the behaviour in combinations
  - `add()` works normally, but not if `remove()` was called before.
  - Maybe only the 2nd call causes an error.
- As far as possible, test only 1 thing at a time.
- Tests should avoid logic as much as possible.
  - No if/else, loops, etc. in the code of the test method.
- Tests should be independent of each other.
  - It should not make any difference whether Test A is run before Test B.

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Unit tests - method

Prof. Dr Silke Lechner-Greite

Contents

Technische Hochschule Rosenheim

- What pattern could we follow for programming unit tests?

Extracted from:

[1] Hunt et al., Unit-Tests mit JUnit, Hanser Verlag, 2004

Which test scenarios are conceivable?

Technische Hochschule Rosenheim

Strategic approach according to the method "Right BICEP" [1]:

Right: does the method work, i.e. does it deliver the right results?

Boundary condition: does the method behave correctly with respect to the boundary conditions? How does the method behave in the so-called boundary areas?

Inverse operation: how does the method behave when I check the reverse operation?

Cross-check: how does the method behave in relation to the cross-check with the result?

Error condition: how does the method behave if error conditions are deliberately tested?

Performance: is the method performance within the expected range?

Right

Technische Hochschule Rosenheim

- The functionality of the method must be correct. This means that the calculation methods and the programme logic must be correct in order to obtain the correct results.

- Aim:
  - » Show that the method behaves correctly.
  - » Show that the method does exactly what is expected of it.

- Basis: requirements for the method: what should the method achieve?

- The sense and purpose of the method must then be verified by means of the tests.

- Considerations:
  - What is the correct test data to show this?
  - What is the simplest test to show that the method works?

Boundary condition

Technische Hochschule Rosenheim

- If a method works “correctly”, it must be checked whether the method also behaves correctly even in so-called boundary areas.

- Aim: show borderline cases for which the method still works correctly.

- Examples:
  - Illegal values are passed
    - e.g. illegal characters in file names: \ / : * ? “ < > |
  - Incorrectly formatted data
    - e.g. wrong e-mail address: smith@th-rosenheim.)
  - Missing values:
    - e.g.: 0, 0.0, “”, null
  - Values outside the permitted range,
    - e.g. age cannot be negative, days of the month, months
  - Duplicated values
    - e.g. Exercise 05: delete a contact based on name, what about duplicate names?
  - (Reverse) Pass sorted list for sorting
  - Reverse the order of the events, i.e. your programme expects a certain sequence logic, what if this is not adhered to?
    - e.g. logout before login

Borderline case – memory aid [1]

Technische Hochschule Rosenheim

Conformance: is the value in the expected format?

Ordering: are the values ordered or not ordered (as applicable)

Range: is the value between MIN and MAX values?

Reference: is the code related to external code (no control over external code!)?

Existence: does the value exist?

Cardinality: are there enough values? Are there too many values? Are there too few values?

Time: does everything happen within the acceptable/prescribed timeframe? At the right time?

Inverse operation

Technische Hochschule Rosenheim

- What is the logically reversed operation?
- This can be executed with some methods.
- At the end, we compare the difference to the expected value, and only accept the test if it is sufficiently low.
- Ideally, we compare our own method with an external method that we did not write ourselves, to avoid any influenced errors.

Cross-check

Technische Hochschule Rosenheim

- Is there an alternative way to calculate the task of the method? For example, this could be another algorithm that calculates something.
- The alternative should be known, ideally a kind of gold standard, or another approach to solving the problem.
- The alternative can be external or self-implemented
  - e.g. working with strings:
  a) Use StringBuilder
  b) Working with strings
  - e.g. reading from scanner:
  a) Use the methods of the primitive data types to parse the result (Integer.parseInt(scanner.nextLine())
  b) Use the scanner methods to parse the result (scanner.nextInt())
- A cross-check is also to check the consistency of the data
  - e.g. we create different objects of the Student class and count them using a counter. Each student is then assigned a state of the counter as a matriculation number. Is the maximum number of student objects equal to the highest matriculation number?

Error condition

Technische Hochschule Rosenheim

- Aim: test how the method / programme handles error situations, and whether the programme continues to function correctly.
- Despite all the tests, there are still situations where errors occur (network errors, time differences, no more disk space, poor graphics resolution, etc.).
- Errors also occur when the method / programme depends on other parts of the programme that are implemented by colleagues which do not yet exist, etc.
- Or the method has unpredictable behaviour, or we would need to initialise a complete programme to test the method
- We provoke error situations through so-called mock objects The test does not use the real object, but an imitated object (not discussed further in this lecture). These mock-ups implement the expected interface of the actual object and replace it.
- Examples:
  - A database should be used in the programme . In testing mode, a mock database is used.
  - Image display on a medical device: use of mock image data.

Performance

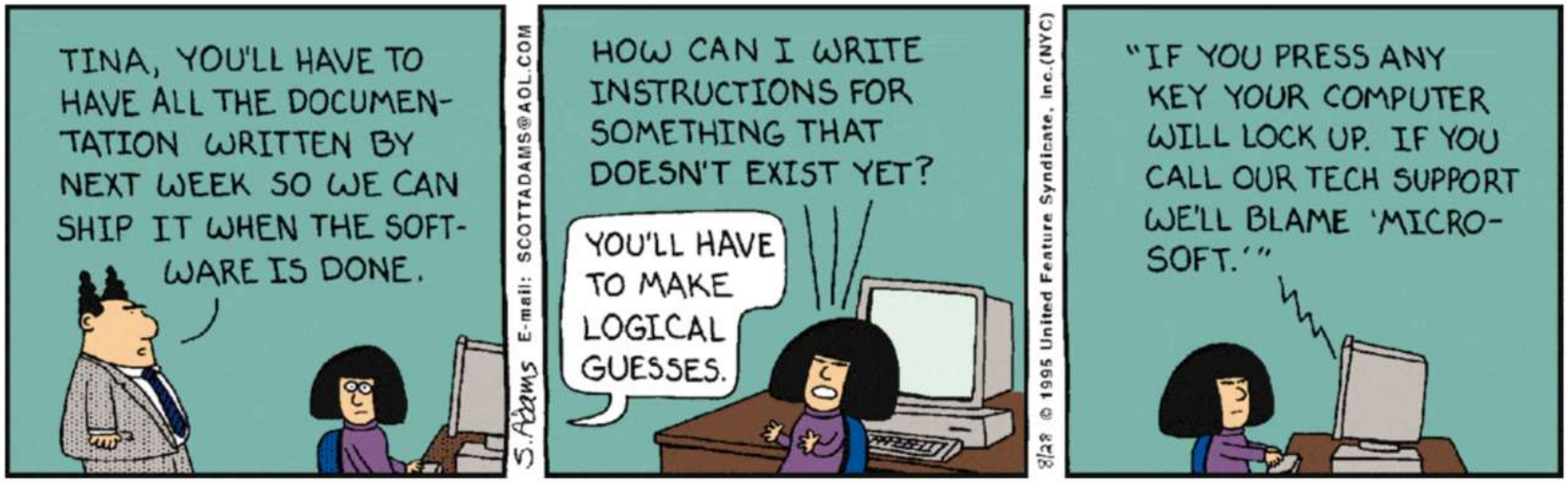
Technische Hochschule Rosenheim

- Aim: demonstration that the programme still performs well in the event of steadily increasing input quantities, a larger database, etc.
- Aim: keep an eye on the performance trend. What changes to the code will cause performance loss?
- Important: traceability, to understand when and where the problem is caused by changes.
- Example: data processing – filtering a text, e.g. for case sensitivity (as with the student's palindrome checker).
  - Test 1: Passing 100 palindromes
  - Test 2: Passing 1,000 palindromes
  - Test 3: Passing 10,000 palindromes, etc.
- This type of test is time consuming. They can be performed at bigger intervals (e.g. every few days, once a week).

Technische Hochschule Rosenheim

Chapter 10: Unit Tests with JUnit, Documentation with Javadoc

For regeneration…



Documentation with Javadoc

Technische Hochschule Rosenheim

- **Motivation**
  - Documentation is often not updated when code is changed.
  - Documentation is often neglected under time pressure.

- **Solution**
  - Integration of source code and documentation, i.e. source code and documentation in the **same** file
  - Extension of the concept of block comments

- **Documentation generator:** `Javadoc`
  - Creates an `.html` file for each `.java` file, with a description of class, interface, methods, etc.
  - Documentation by means of special comments
    - Are contained in the source code immediately before the section to be documented
    - Start with `/**` and end with `*/`
    - Can consist of multiple lines
    - The first sentence (up to the first dot) is a short description

Javadoc example

Technische Hochschule Rosenheim

```
/**
 * Simple implementation for rational numbers.
 *
 * Rational numbers are represented by numerators and denominators.
 * @author Computer Science professors
 * @version 1.1
 */
public class Rational {
   private long numerator;
   private long denominator;
   /**
    * Rational number with numerator and denominator of the type long
    * @param num numerator
    * @param den denominator
    */
   public Rational(long num, long den) {}
   /**
    * Adds two rational numbers.
    * @param val rational number which should be added up to.
    * @return A new rational number as the result of the operation
    */
   public Rational add(Rational val) {
      return null;       // just to shorten the code
   }
}
```

Javadoc for class declaration

Javadoc for constructor

Javadoc for method

Structure of a Javadoc comment

Technische Hochschule Rosenheim

- Documentation of
  - Classes and interfaces
  - Methods
  - Attributes (data elements)
- Contents of Javadoc comments
  - Description (summary and details)
  - Tags →highlight key information

- Tags
  - Structure: *@keyword [parameter] text*
    - *keyword* refers to key information
    - *text* stands for continuous text
  - Different tags for
    - Classes and interfaces
    - Methods
  - No tags for data elements
  - Javadoc tags are not annotations

Javadoc: the most important tags

Technische Hochschule Rosenheim

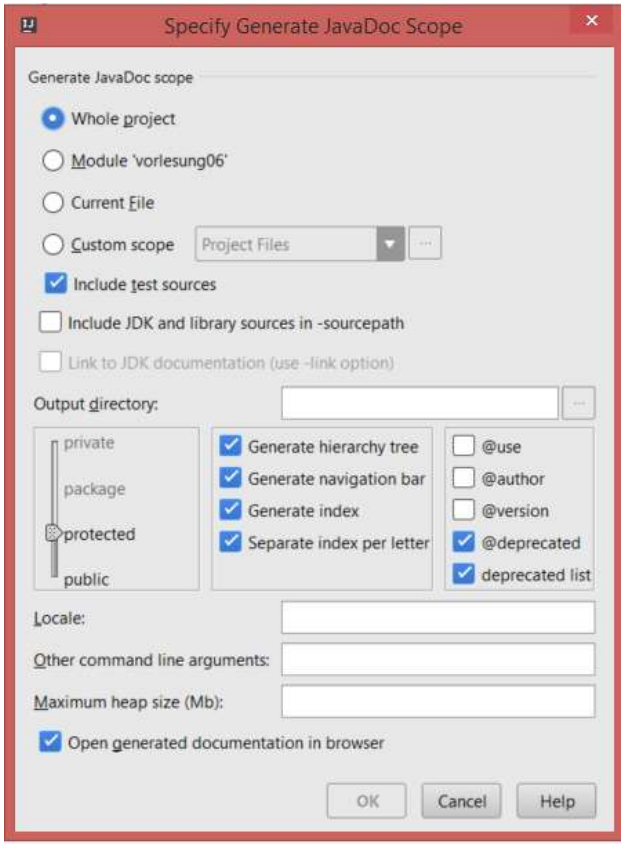
- Tags for **classes and interfaces**
  - *@author text*
    - Name of the author / authors
  - *@version text*
    - Version of the source code

- Tags for **methods**
  - *@param name text*
    - Meaning of the parameter *name*
    - Repeated for each parameter
  - *@return text*
    - Meaning of the result of the method
    - Missing for `void` methods and constructors
  - *@throws exceptionclass text*
    - Notice of any exception class thrown (see later)
    - Repeated for each exception

Creating the Javadoc documentation

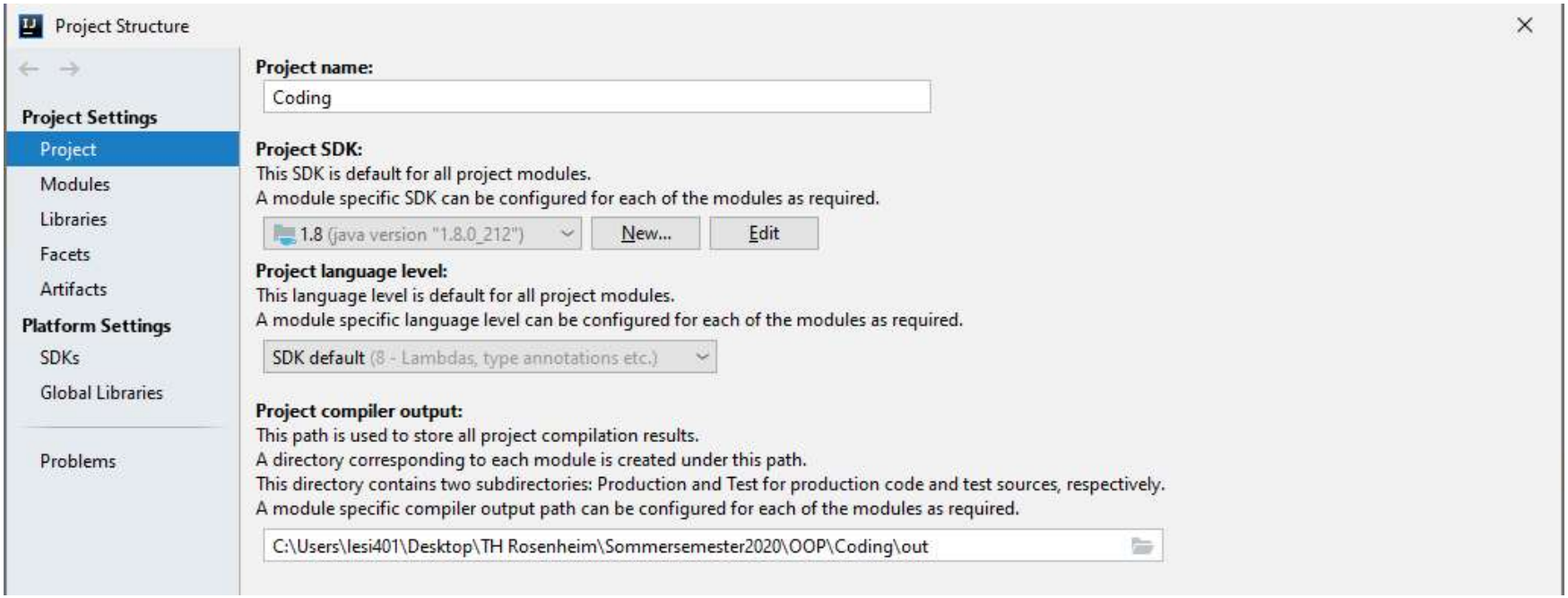
Technische Hochschule Rosenheim

- Special compiler `javadoc` is part of the JDK
  - Can be called via the command line.
- Result readable with any web browser
  - One HTML page per class
- IntelliJ
  - *Tools → Generate Javadoc*



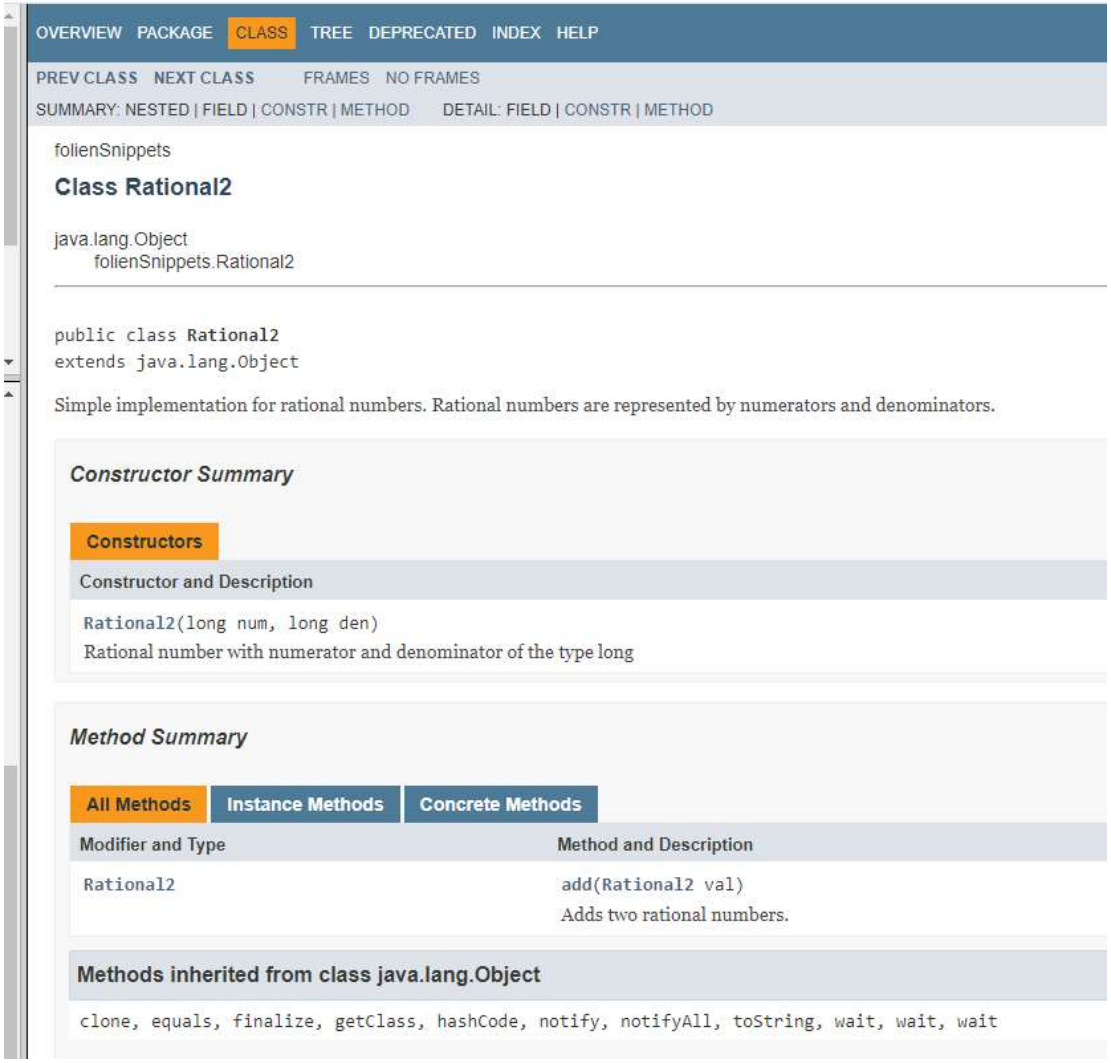
If Javadoc doesn't run in IntelliJ:

- Open Project Structure Ctrl+Alt+Shift+S (File -> Project Structure)
- Under Platform Settings – SDK - + - add the path where JDK is installed
- Under Project – then select it



Creating Javadoc documentation

Technische Hochschule Rosenheim



OVERVIEW PACKAGE CLASS TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

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

folienSnippets

Class Rational2

java.lang.Object
  folienSnippets.Rational2

```
public class Rational2
extends java.lang.Object
```

Simple implementation for rational numbers. Rational numbers are represented by numerators and denominators.

Constructor Summary

Constructors

Constructor and Description
`Rational2(long num, long den)` Rational number with numerator and denominator of the type long

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
`Rational2`	`add(Rational2 val)` Adds two rational numbers.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Summary

Technische Hochschule Rosenheim

- Test-driven development
  - Write tests first, then implement!
  - Improves quality of software
- Annotations
  - Store meta information in the programme code
- JUnit 5
  - Library for easy creation and execution of unit tests in Java.
- Javadoc
  - Documentation of a programme within the code.