Tests

Quality assurance in software development

How did you handle it so far?

Unit Tests

with JUnit

AW Academy

Unit Tests

- Software, wich checks my software
- Tests smalles possible *units* of the software: single class or method

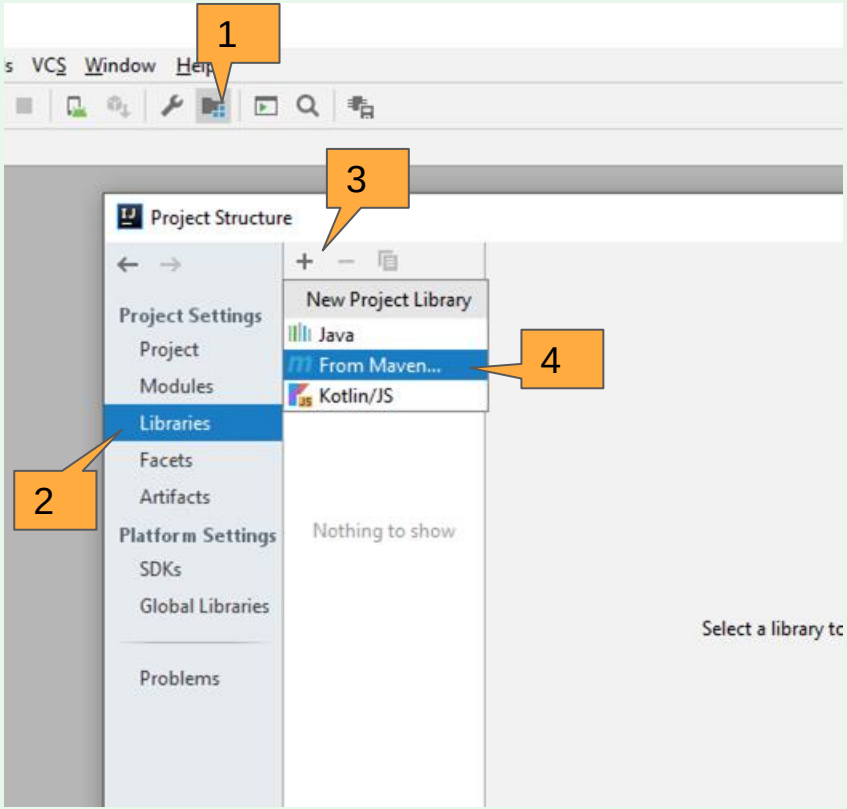
Test profekt: Calculator (Prestudy W4)

- Open the project
- Calculator class still there?

```java
public class Calculator {
   private int result = 0;
   public int getResult() {
       return result;
   }
   public void plus(int zahl) {
       this.result += zahl;
   }
   public void minus(int zahl) {
       this.result -= zahl;
   }
   public void times(int zahl) {
       this.result *= zahl;
   }
   public void divided(int zahl) {
       this.result /= zahl;
   }
}
```

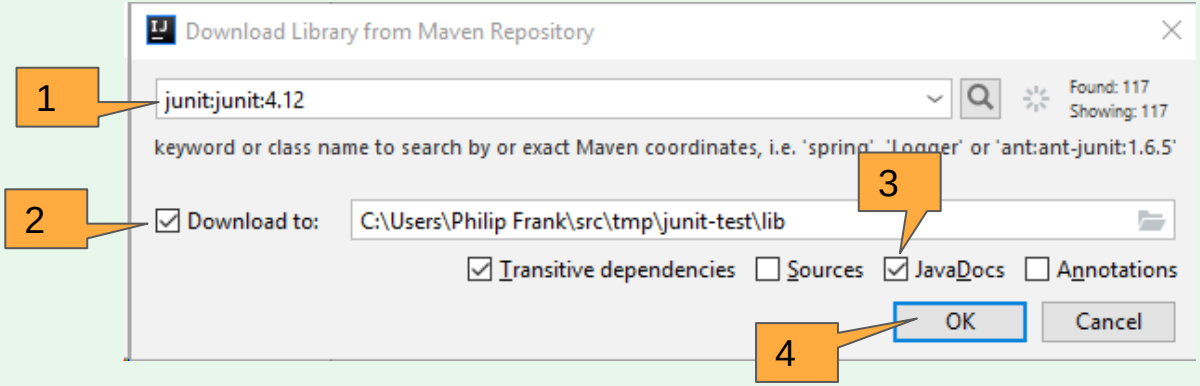
AW Academy

Adding JUnit to a project

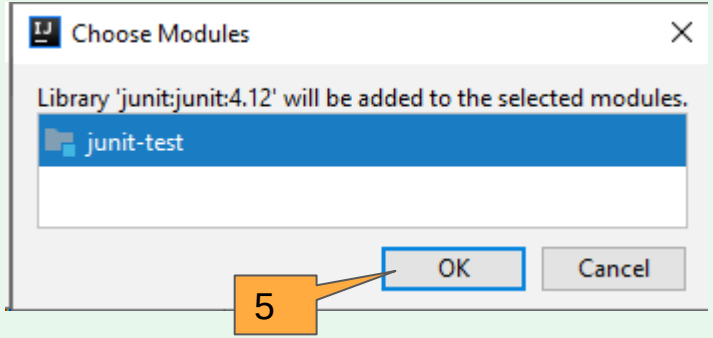


Academy

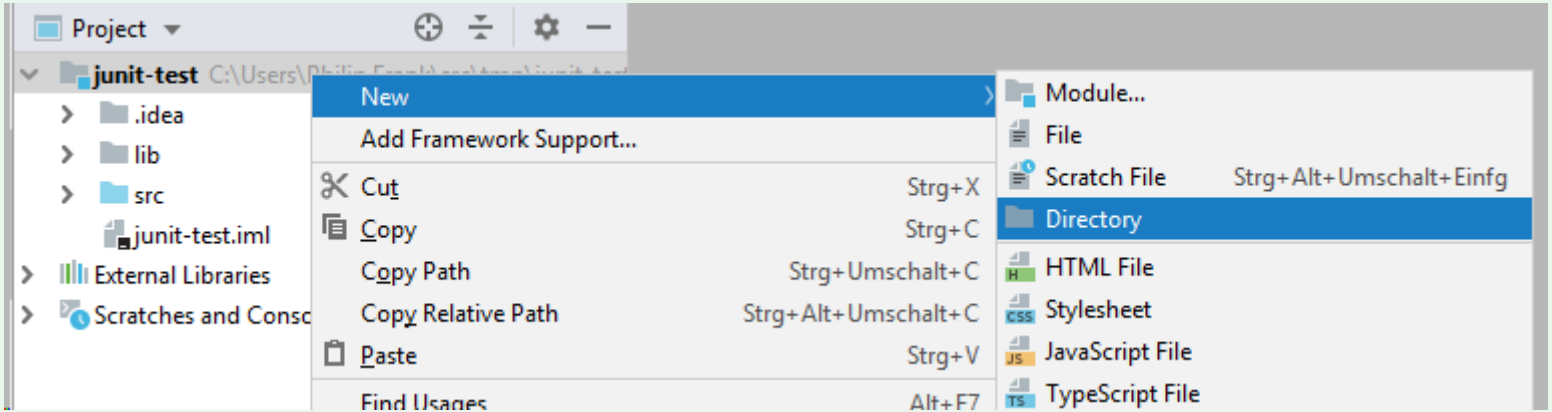
Adding JUnit to a project

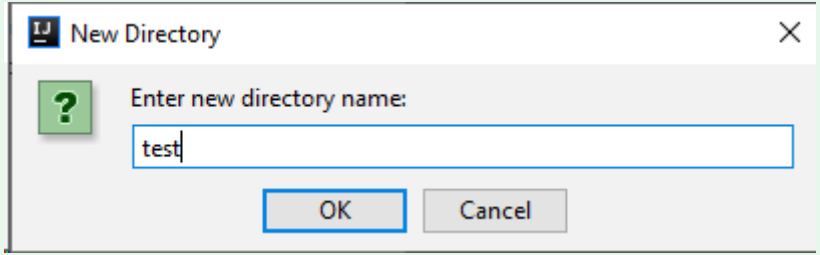



```
junit:junit:4.13
```



Create test directory

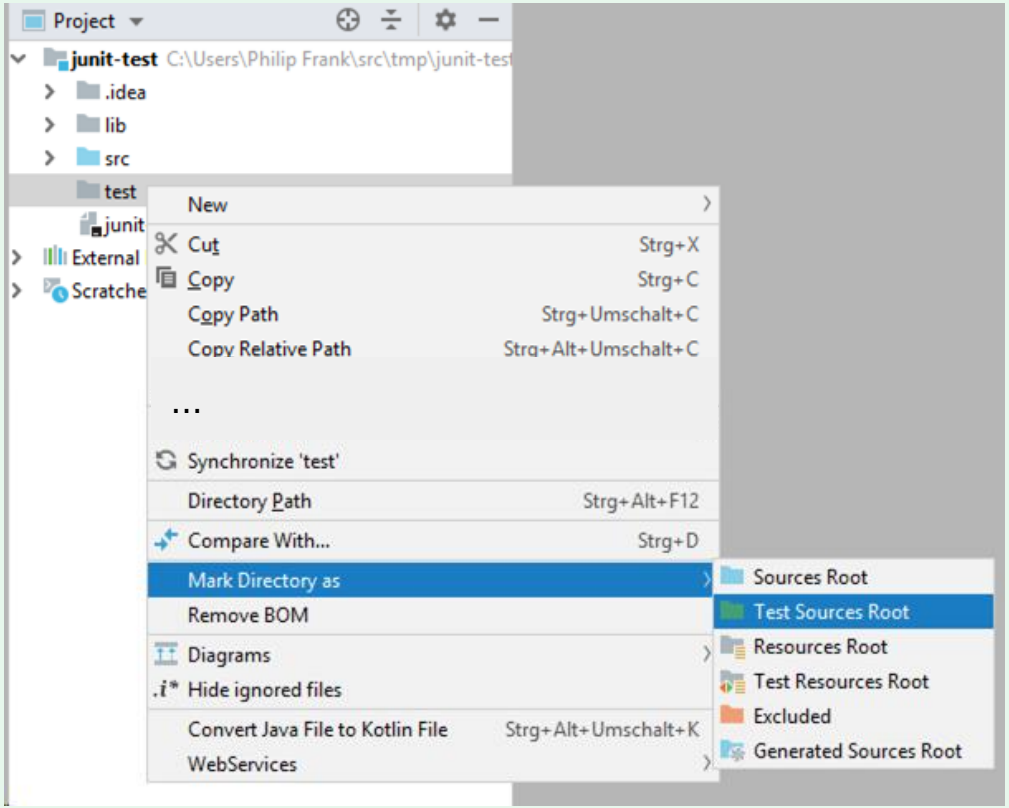




Important: Start with a right click on the name of the project, to make the directory on the topmost level.
It should **not** be a subdirectory of `src`, `lib` or `.idea`

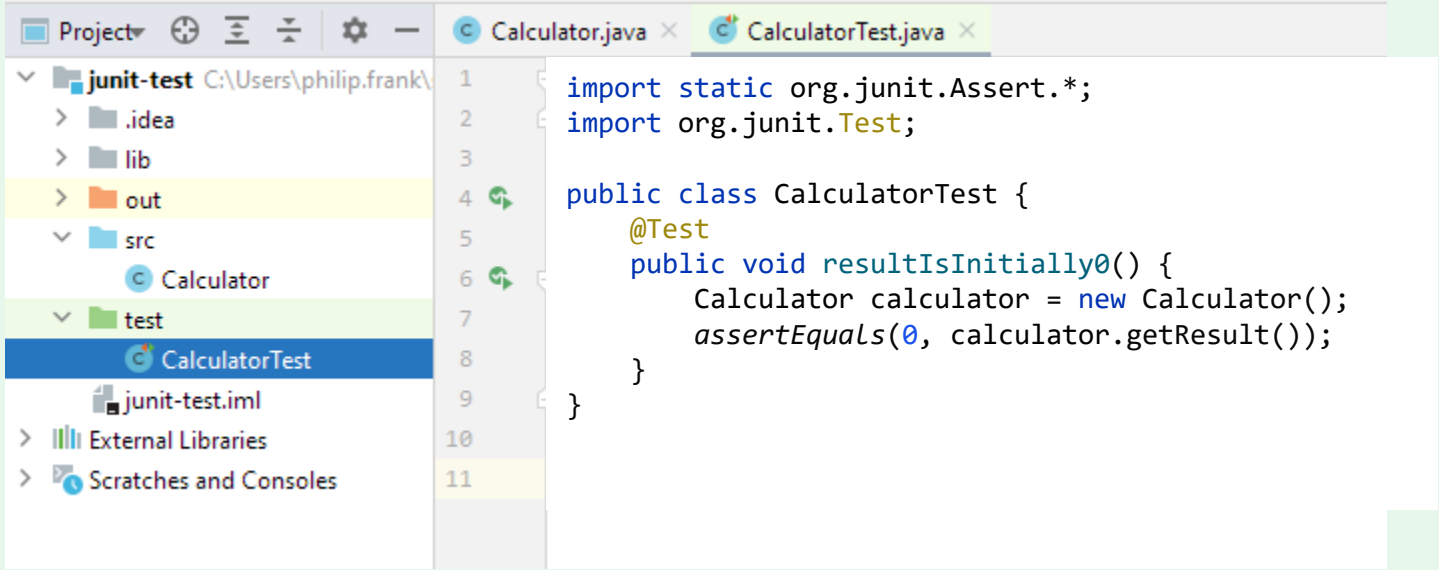
AW Academy

Mark test directory as such



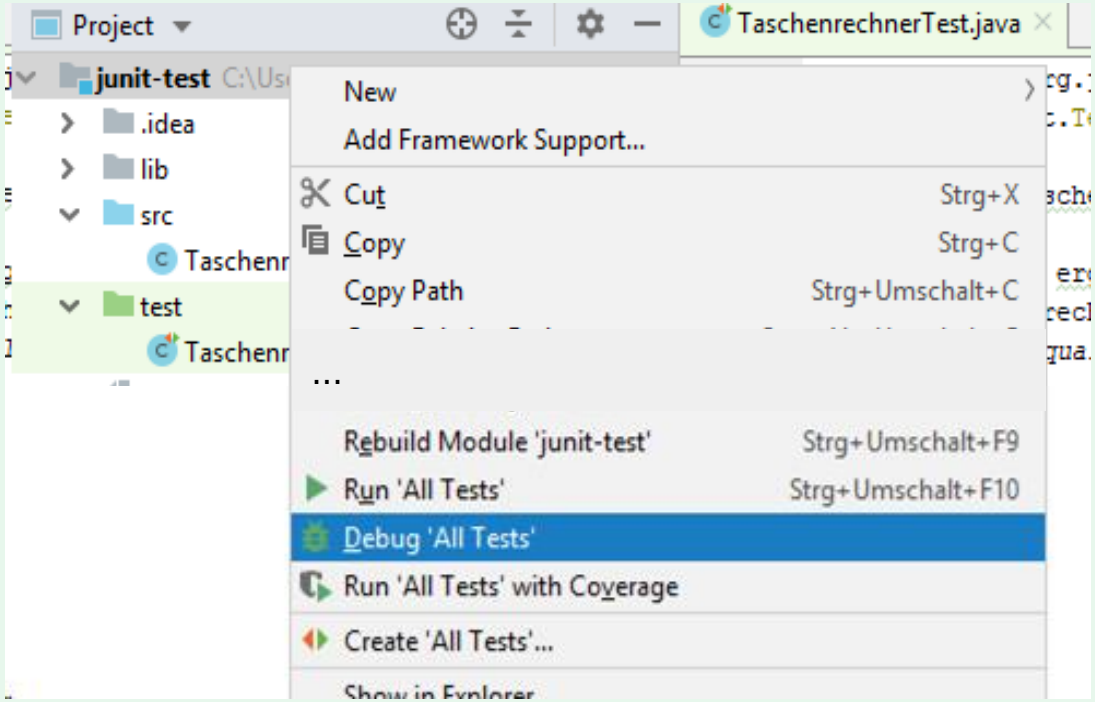
AW Academy

Create Test class and first test



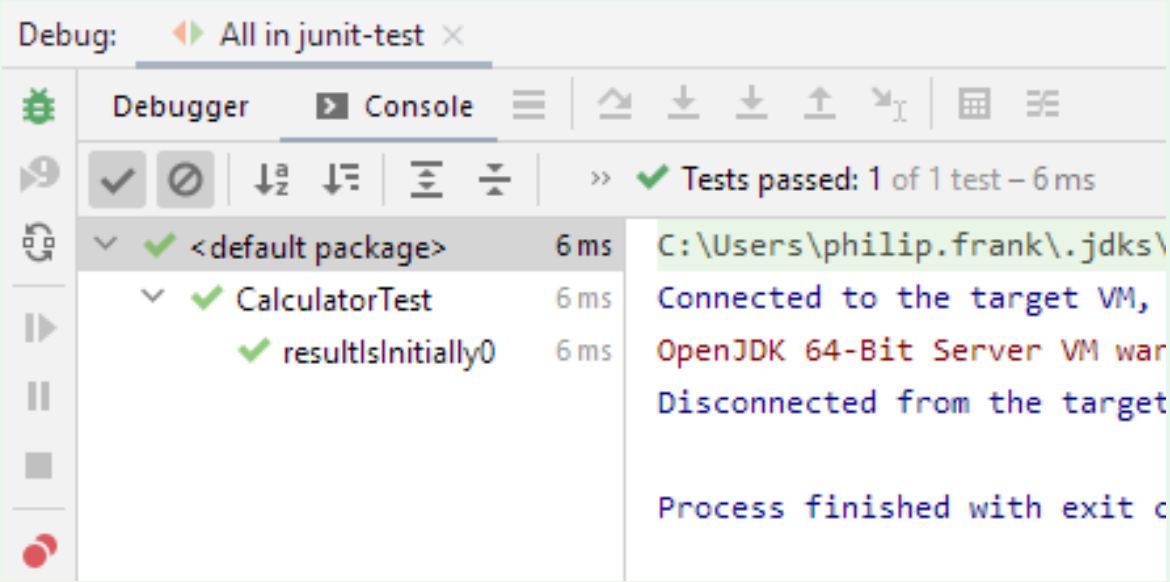
Academy

Run Test



Academy

Success!



Academy

Test Class in Detail

`import static` for assert methods
saves us from writing **`Assert.`** every time

`@Test` marks a method as test

Project

Calculator.java | CalculatorTest.java

- lib
- out
- src
  - Calculator
- test
  - CalculatorTest
- Scratches and Consoles

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

public class CalculatorTest {
    @Test
    public void resultIsInitially0() {
        Calculator calculator = new Calculator();
        assertEquals(0, calculator.getResult());
    }
}
```

Test classes go in here

Usage of an assert method

Academy

Assertions

Assert methods

- make assertions that are checked for validity.
- the test fails, if the assertion is not valid
- name starts with „**assert**“, z.B. assertEquals
- assertEquals: two parameters **expected** and **actual** (in this order)

```
assertEquals(0, calculator.getResult());
```

- Optional: Failure description as leading string parameter:

```
assertEquals("Initial result incorrect", 0, calculator.getResult());
```

`assertTrue(condition)`

would be enough, to build all other assertion methods.

many more „c**onvenience methods**“ are available

Assert methods

Boolean

```
assertTrue
assertFalse
```

Object Checks

```
assertEquals(Object, Object)
assertSame(Object, Object)
```

Null Checks

```
assertNotNull
assertNull
```

Primitive Checks

```
assertEquals(long, long)
assertEquals(double, double)
```

Arrays

```
assertArrayEquals(Object[])
assertArrayEquals(long[])
```

```
fail()
```

Extensive test for the calculator

Write tests and execute them right away:

1. Simple tests for all four basic calculations
2. A longer test, combining all calculation types
3. A test with numbers greater than 10000
4. A test with negative number as result
5. A test with negative number as parameter, e.g. **`plus(-4)`**
6. Now break the calculator on purpose: Make one method behave incorrecty, e.g. always +1 the result.
    1. How is the error noticeable?
    2. How can you re-run only the failed tests?

Bonus: Test with result greater than 3 billion, Test division by **`0`**

AW Academy

Changing the implementation of a method

Change the implementation of **`times()`** so it does not use the **`*`** operator, but instead uses a loop of additions.

The result should still be correct and pass all tests. Make sure it also works for **`0`**, **`1`** and negative values.

Bonus: Write tests for **`divided()`**, where rounding of **`int`**s happens. Now replace the implementation of **`divided()`** with a loop of subtractions.

Improving calculator

Without changing the existing tests, and with matching new tests added:

1. New method **`clear()`**, which sets the result to **`0`**
2. Additional constructor with one parameter for the initial result.
3. Method **`absolute()`**, which sets the result to its positive (absolute) value.
   (e.g. Absolute -5 is 5; Absolute 3 is 3)

Bonus:

1. Method **`power()`**, which exponentiated the result
2. Method **`rount()`**, which rounds the result to a given number of significant digits

Review and preview

- Why are tests useful?
- Where are tests stored in a project?
  - What if I have a test for multiple classes?
- Assert methods
  - Which ones were useful so far?
  - Order of arguments?
- Implementation first, then write tests?

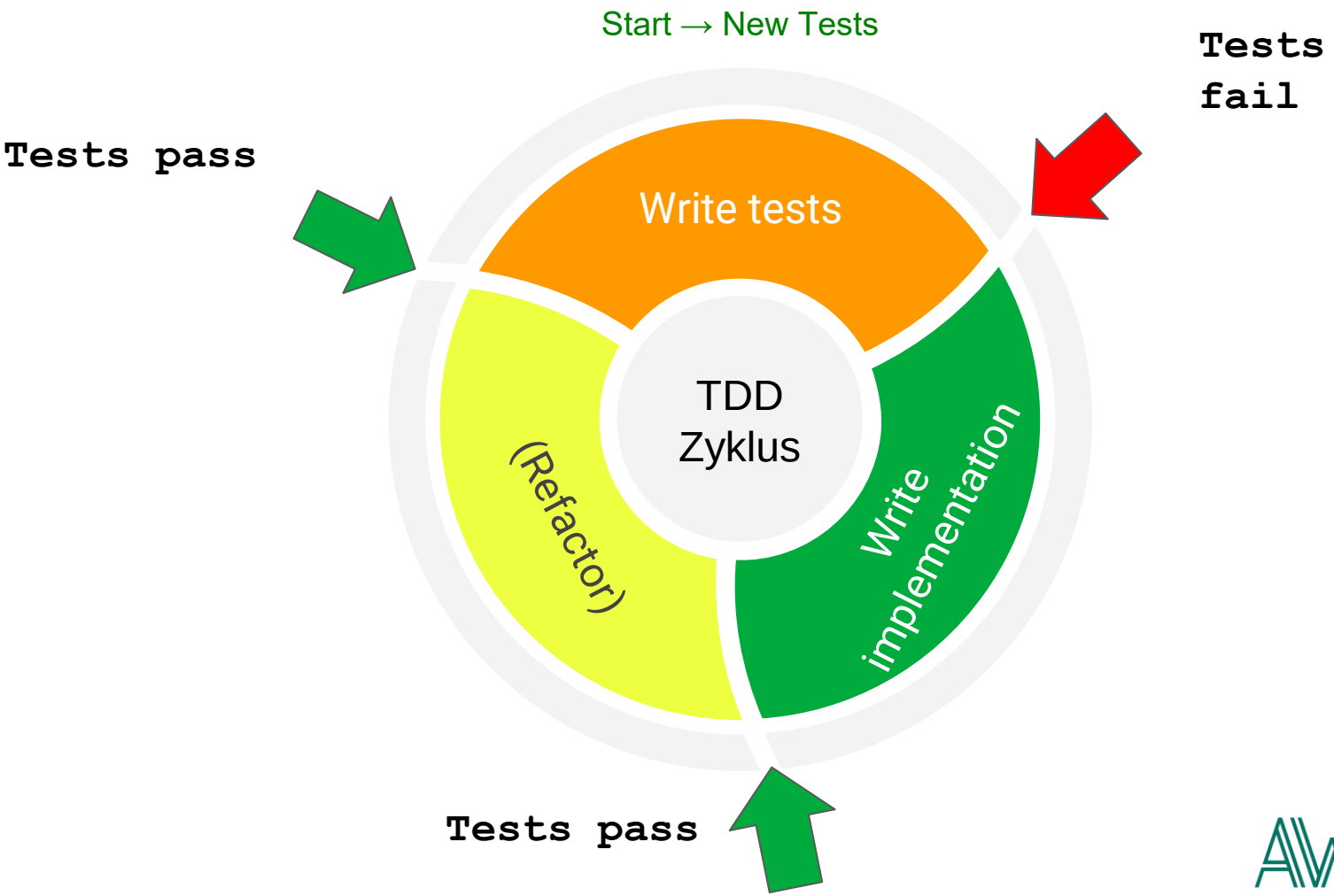
Test-driven Development

(TDD)

AW Academy

Test driven development

With test-driven development, the tests are written first, which then guide the programming.


Start → New Tests
Tests fail
Tests pass
Write tests
TDD Zyklus
(Refactor)
Write implementation
Tests pass

AW Academy

Cyclic flow

Programming takes place in three phases.

1. Tests are written that initially fail.
2. Exactly as much (!) Code is written as is necessary to pass the tests successfully.
3. The code and tests are refactored.

Incremental approac

Test-driven development describes incremental programming and is an evolutionary procedure.

- Each TDD cycle adds a new functionality to the software.
- The duration of a cycle is in the range of minutes, maximum hours.

Design strategy

Properties of test-driven development:

- TDD is a design strategy for the software development process, not a test method.
- Test-First: Test cases are written before (!) The implementation of the logic.
- Classes and methods are used before they are fully implemented.
- Refactoring: Design simple, redundancy-free software that is easy to understand.

TDD cycle

The 3 phases of the TDD cycle do not overlap.

⇒ Each activity can be clearly assigned to a phase.

E.g. ...

- ... tests are only written in the “Write Tests” phase.
- … Tests and code are only simplified in the 3rd phase (“Refactoring tests and code”).
- ... no tests were written in phase 2 (“writing code”).

TDD Exercise: Cash register

Product has name and price (in Cent as **`int`**)

Products can be scanned by Register

Register calculates subtotal of scanned products.

Template without real implementation:

```java
public class Product {
    public Product(String name, int price) { }

    public String getName() {
        return null;
    }

    public int getPrice() {
        return 0;
    }
}
```

```java
public class Register {
    public void scan(Product product) { }

    public int getSubtotal() {
        return 0;
    }
}
```

AW Academy

TDD Exercise: Cash register

Perform a TDD cycle for each step:

1. Product constructed with name and price

   Test examples: Correct value from **`getName()`**, **`getPrice()`**
2. Product is scanned by register

   Test examples: Correct value from **`getSubtotal()`**
3. Multiple products scanned by register

   Test examples: Correct values from **`getSubtotal()`** after every scan

TDD Exercise: Cash register, extended

Perform a TDD cycle for each step; all existing tests must remain unchanged!

1. Add method **`pay()`** to cash register: Returns the amount to pay and resets it, so it's the next customer's turn.

   Test example: Correct amount returned, afterwards correct value from **`getSubtotal()`**
2. Add method **`pay(int paidAmount)`** : Like above, but returns change amount for the customer

   Teste example: Correct return value
3. Register handles creit (e.g. from returned bottle deposit)
4. Discount voucher: 10%, also applies to all of the following customer products
5. Cash register can cancel the last scanned product once.

TDD Exercise: Cash register, Bonus

1. Some products are part of a loyalty program. When purchasing loyalty products worth at least € 10 (in one purchase), the customer receives a 5% discount on the purchase. Also note special cases with credits for other discounts.
2. The management would like to know at the end of the day:
    1. How many customers have shopped
    2. How much turnover was made
    3. How much turnover was generated per purchase on average
    4. How much discount was granted through the 10% discount and loyalty program
    5. What percentage of customers shopped for more than € 100

AW Academy