

JUNIT TEIL 2

... schön, dass Sie wieder
eingeflogen sind



Wo standen wir?



Begriffe vom ersten Teil

ExecutionCondition
grüne Tests

Teststruktur
Testinput

@AfterAll

JUNIT 5

@BeforeAll
@Test

NestedTest

@AfterEach

Jupiter

rote Tests

Qualität

@ExtendWith
@Disabled

Assumptions

Test managen

Testen allgemein

AAA-Pattern

Testarten

Assertions

@BeforeEach

Hier noch mal: Ressourcen

› Golden-Source:

- GIT-HUB:
- Diese Unterlage
- Beispiel-Projekte
- Aufgaben-Projekte

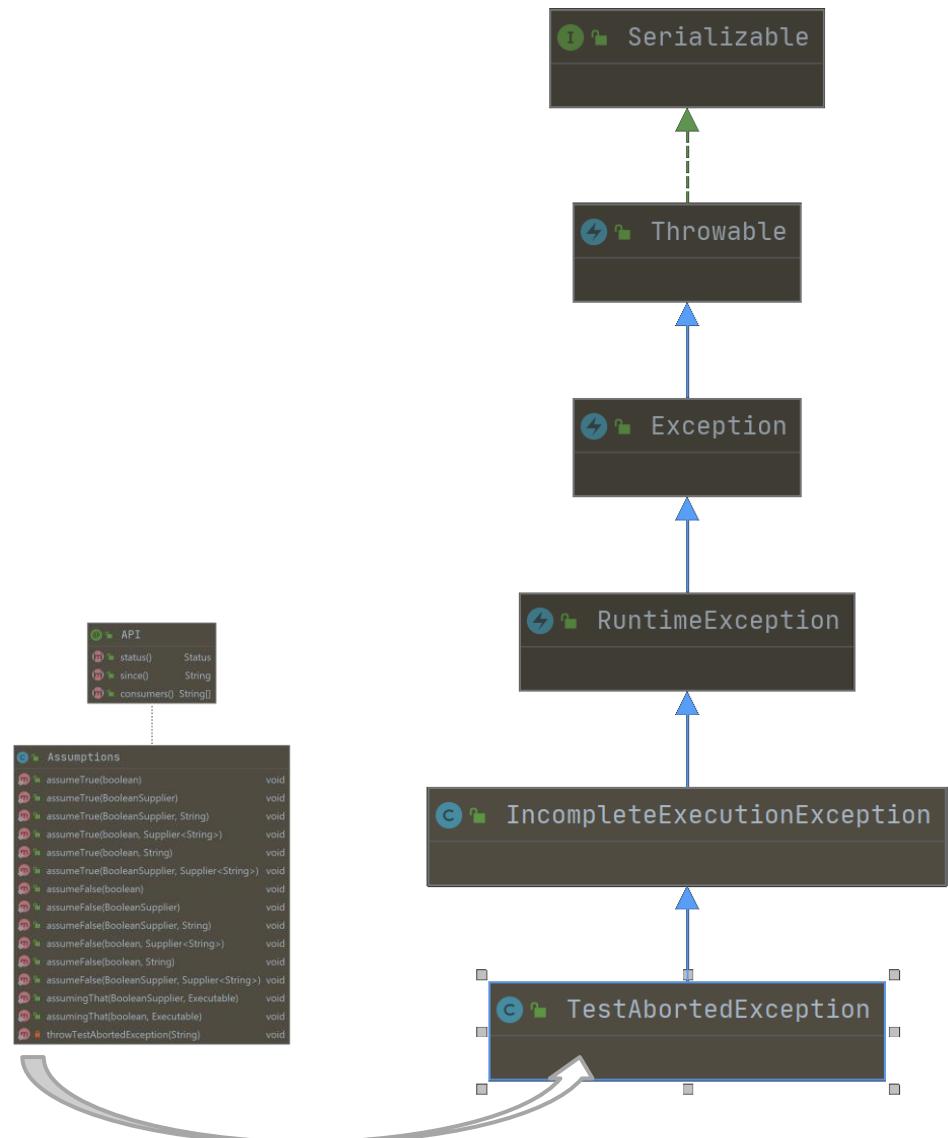
A photograph of two brown bears swimming in a river. One bear is facing right, and the other is facing left, seemingly interacting or competing for a fish. The water is brown and rippled.

Gibt's noch was?

- Was wäre heute für mich wichtig?
 - Was sollte heute nicht fehlen?
 - Was sollte heute besser laufen?

Zusicherung, die zu Abbruch ohne Fehler führt: Assumptions

- › Hierdurch können Vorbedingungen zugesichert werden
 - › Vereinfachung von Teststrukturen



Achtung bei Assumptions

```
public class AssumptionTest {  
  
    @Test  
    public void disabledWithFollowingCode() {  
        Assumptions.assumeTrue( assumption: false, message: "assume is working");  
        // not proceed  
        Assertions.assertTrue( condition: false, message: "this will not fail ;)");  
    }  
  
    @Test  
    @Disabled  
    public void disabledWithoutProceedingAnyCode() {  
        Assertions.assertTrue( condition: false); // complete test is disabled  
    }  
  
    @Test  
    public void disabledByAssumingWithImportantTest() {  
        int anyFirst = 1;  
        int anySecond = 2;  
        Integer sum = add(anyFirst, anySecond);  
        Assumptions.assumeTrue( assumption: sum!=null);  
        Assertions.assertEquals( expected: 3, sum);  
    }  
  
    // only for training placed here  
    private Integer add(Integer firstSummand, Integer secondSummand) {  
        return null;  
    }  
}
```

- Wirkung ist ähnlich ausgeschalteter Tests (vielleicht sogar schlimmer, Ausschalten geschieht programmatisch ...)
- Immer gut überlegen, ob ein Test nicht besser rot werden sollte

▼	⌚ AssumptionTest	14 ms
⌚	disabledWithFollowingCode()	13 ms
⌚	disabledByAssumingWithImportantTest()	1 ms
⌚	disabledWithoutProceedingAnyCode()	

Allgemeines Ausschalten von Tests

- › Tests können ausgeschaltet (disabled) werden
 - › Annotation: `@org.junit.jupiter.api.Disabled`
 - › Mögliche Angabe einer Dokumentation (Grund, Task, ...)
 - › Möglich an Testmethode oder Testklasse



Vorsicht bei allgemein ausgeschalteten Tests

- › Ausgeschaltete Tests fallen nicht immer sofort auf
- › Ausgeschaltete Tests verursachen keinen Fehler
- › Ausgeschaltete Tests bleiben häufig versehentlich ausgeschaltet
- › Jeder Test hat seinen Sinn
 - › Falls nicht, Test löschen und nicht disablen
 - › Bei einem Problem lieber sofort lösen
- › Niemals einen roten Test disablen



Ausschalten zur schnellen Arbeit

- › Während der Entwicklung macht es mitunter Sinn, langsame Tests auszublenden
- › Aber bei Abschluss sollten dann wieder alle Tests ausgeführt werden

```
@Test
@Disabled
public void disabledTestWithoutReason() {

}

@Test
@Disabled("the test goes red, but I don't know the reason ...")
public void disabledTestBySillyReason() {

}
```

Allgemeines Ausschalten von Tests

- › Tests können ausgeschaltet (disabled) werden
 - › Annotation: `@org.junit.jupiter.api.Disabled`
 - › Mögliche Angabe einer Dokumentation (Grund, Task, ...)
 - › Möglich an Testmethode oder Testklasse

Idealerweise eine Lösung/Issue mit angeben

```
@Test  
@DisplayName("will be solved by issue #12")  
public void domainTest() {  
  
}
```



Ausschalten zur schnellen Arbeit

- › Während der Entwicklung macht es mitunter Sinn, langsame Tests auszublenden
- › Aber bei Abschluss sollten dann wieder alle Tests ausgeführt werden

Bedingtes Ausschalten von Tests

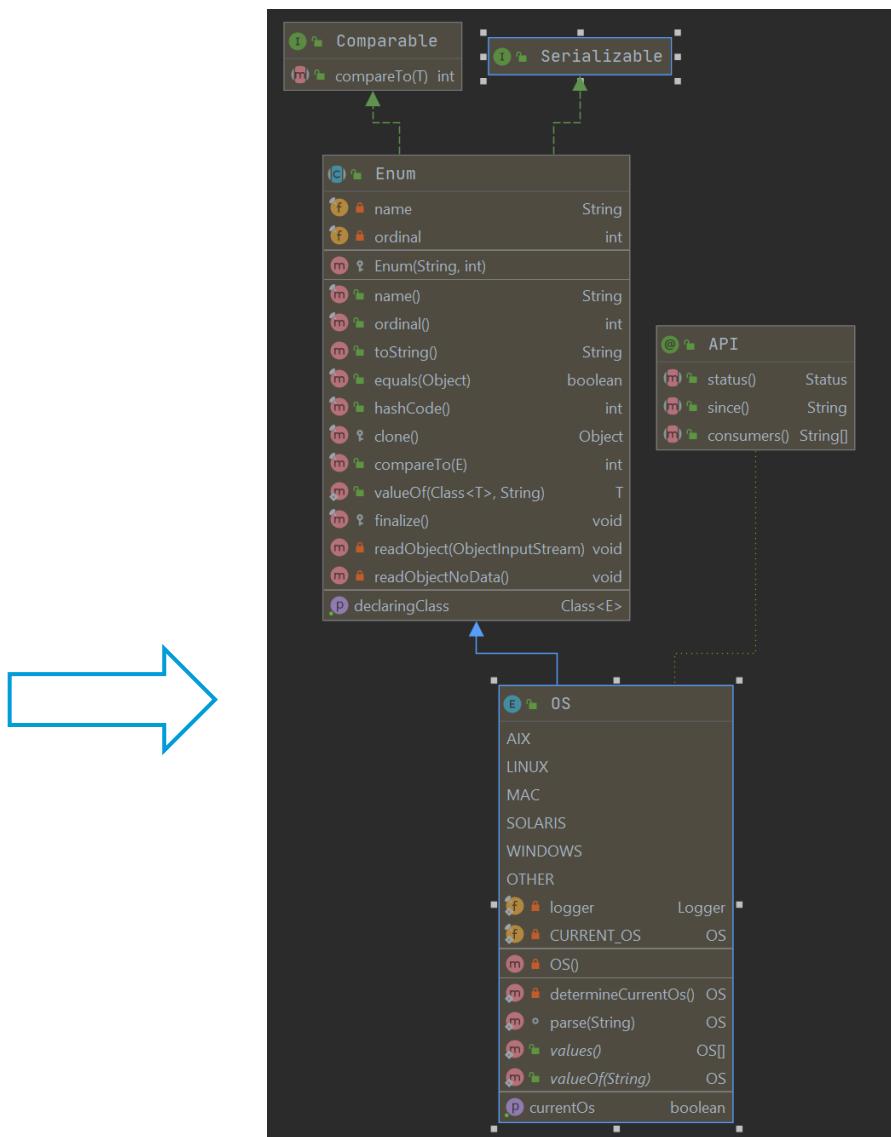
- › Tests können abhängig vom Vorliegen/nicht Vorliegen bestimmter Eigenschaften ein/ausgeschaltet werden
 - › Betriebssystem-Variante
 - › JRE-Version
 - › System-Variable(n)
 - › Komplett eigene Bedingungen
 - › Bedingungen aus Script/Methoden
- › Tests können auch unterteilt werden
 - › Gradle/Maven: Unterscheidung „Komponenten/Integrationstest“
 - › Auch sonst erweiterbar über Kategorien

```
@Test
@DisabledIfSystemProperty(named = "os.version", matches = ".*10.*")
void notRunOnlyOnWindows10() {
    System.out.println("not run this only on windows 10 version");
}

@Test
@EnabledIfSystemProperty(named = "os.version", matches = ".*10.*")
void runOnlyOnWindows10() {
    System.out.println("Run this only on WINDOWS OS 10 version");
}
```

Bedingte Ausführung: Abhängig vom Betriebssystem

→ @EnableOnOs(...)
@DisableOnOs(...)



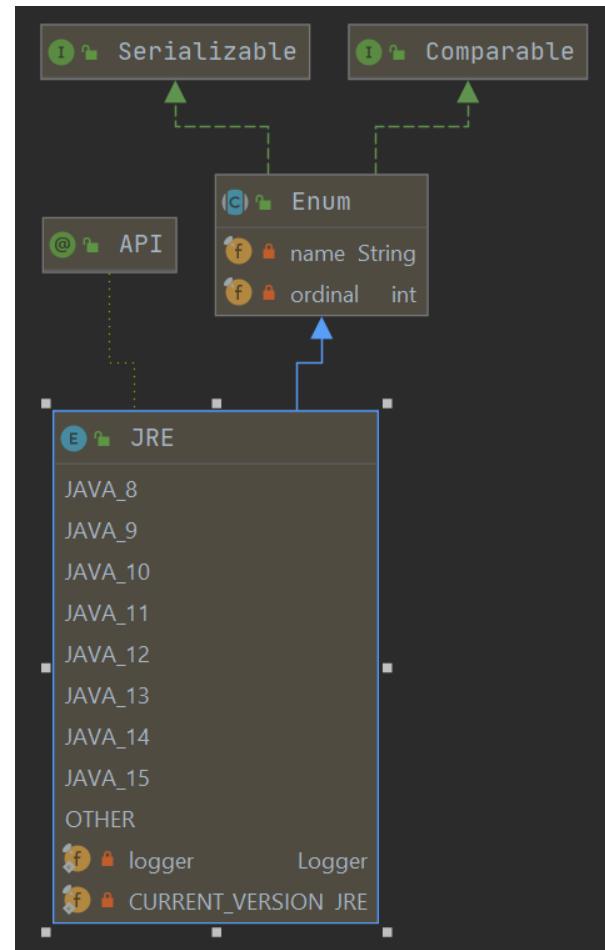
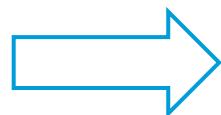
Bedingte Ausführung: Abhängig von JAVA-Plattform

@EnableOnJre(...)
@DisableOnJre(...)

@EnableForJreRange(min=..., max=...)
@DisableForJreRange(min=..., max=...)

@EnableForJreRange(min=...)
@DisableForJreRange(min=....)

@EnableForJreRange(max=...)
@DisableForJreRange(max=....)



Bedingte Ausführung: Abhängig von (JAVA)-System-Property

→ @EnabledIfSystemProperty(named=..., matches=...)

→ @DisabledIfSystemProperty(named=..., matches=...)

→ System.getProperties()

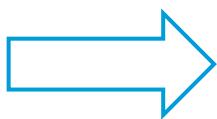
→ System.getProperty(...)

Bedingte Ausführung: Abhängig von (OS)-Environment



@EnabledIfEnvironmentVariable(named=..., matches=...)

@ DisabledIfEnvironmentVariable(named=..., matches=...)



Hintergrund zu bedingten Ausführung

- › Basis ist org.junit.jupiter.api.extension.ExecutionCondition
- › Erstellung einer Annotation
 - › Mit „@ExtendWith“ der erstellten ExecutionCondition

```
public class HogiSampleTestCondition implements ExecutionCondition {  
  
    @Override  
    public ConditionEvaluationResult evaluateExecutionCondition(ExtensionContext extensionContext) {  
        Optional<AnnotatedElement> element = extensionContext.getElement();  
  
        if (element.isPresent()) {  
            HogiCondition annotation = element.get().getAnnotation(HogiCondition.class);  
            if (annotation != null) {  
                String info = annotation.info();  
                return ConditionEvaluationResult.disabled("don't have time for proceeding this boring test: " + info);  
            }  
        }  
        return ConditionEvaluationResult.enabled(...);  
    }  
}  
  
{@Target({ElementType.TYPE, ElementType.METHOD})}  
@Retention(RetentionPolicy.RUNTIME)  
@ExtendWith(HogiSampleTestCondition.class)  
public @interface HogiCondition {  
    String info() default "";  
}
```



Kleiner Rückblick

- › Für eine Test-Methode: @Test-Annotation (auf Jupiter-Test achten)
 - › Für Ausführung in Maven/Gradle auf Dateipattern achten
- › Assertions-Klasse enthält diverse Zusicherungsprüfungen
- › Test-Styleguide zu Eigen machen
- › Implementieren Sie Tests gemäß des AAA-Pattern
- › Möglichst einen guten Testumfang finden
 - › 100% Abdeckung mit möglichst wenig Testfälle



Übung 07.1

› Resourcen

de.hegmanns.training.junit5.practice.task07

a)

Erstellen Sie eine Ausführungskondition, die mit einer bestimmten Annotation (@DisableOnTime) und Angabe eines Zeitraums (@DisableOnTime(from=„12:00“, until=„12:10“)) versehene Testmethoden disabled.

Es ist bereits eine entsprechende Test-Klasse vorbereitet mit zwei Methoden.

b)

Verknüpfen Sie die soeben erstellte Ausführungskondition mit der Environment-Eigenschaft „ENV“, so dass die markierten Tests nur in einem bestimmten Environment nicht ausgeführt werden.
(beispielsweise auf einem build-Server)

Hinweis:

So etwas kann sinnvoll sein, wenn es beispielsweise bei Integrationstest Zeiten gibt, in denen bestimmte Services (Datenbanken, etc) entweder nicht verfügbar sind, oder durch andere Tests stark blockiert sind.



Übung 07.2

→ Resourcen

KEINE

Erstellen Sie die Lösungsklassen im Package
de.hegmanns.training.junit5.practice.task07

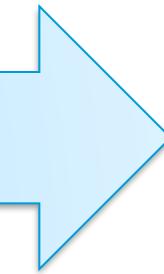
Erstellen Sie eine Ausführungskondition, die abhängig vom Methodenname eine Methode nicht ausführt.
Sofern entweder im Anzeigename oder im Methodenname der Wortbestandteil „long“ vorkommt,
soll die Methode in Abhängigkeit der Property „no.long.tests=true“ gefiltert werden.

Es ist bereits eine entsprechende Test-Klasse vorbereitet mit zwei Methoden.

Tests ausfiltern durch Kennzeichnungen

Junit 4
@Category

@Tag
@Tags

- 
- › Tag ist eine Möglichkeit der Segmentierung von Test-Cases
 - › Ähnlichkeit zu TestSuite (TestSuite ist aber auf Klassenbasis)
 - › Einfachere Anwendung in JUnit5
 - › Nur noch eine Text-Repräsentation ohne Notwendigkeit einer Klasse
 - › Erstellung fester Annotationen als Sub-Annotationen möglich
 - › Verwendbar an Klasse und Test
 - › Entspricht quasi einem logischen „ODER“
 - › Gleiches Konzept in Build-Engines (Maven, Gradle)
 - › Hinzunehmen von Tags
 - › Ausschließen von Tags

Kategorisieren : @Tag

- › Grundsätzliche Kategorisierung: @Tag
 - › Einfacher als in JUnit4
(ähnlich zu @Category, aber mit eigenen Klassen)
- › Muss genau eine Bezeichnung enthalten
- › Auf Basis von @Tag eigene Annotationen möglich
- › Möglich an Klasse und Methode
 - › Auch mehrmals
- › Annotationen an Klasse werden in Subklassen vererbt
- › Kann feingranular in Maven / Gradle angesprochen werden



Regeln / Syntax für Tag-Namen

Keine leere Zeichenkette, keine Whitespaces, nicht „null“

Keine Kontroll-Sequenzen

Keine reservierten Zeichen:
Komma (,), Runde Klammern (), Kaufmannsund (&), Bar (|), Ausrufezeichen (!)

Kategorisieren : @Tag Aus/Einschließen

- › @Tag-Annotation
 - › Klasse: Tag gilt für alle Testmethoden
 - › Methode: Tag gilt für diese Methode
- › Tag einschließen
- › Tag ausschließen
 - › Sinnvoll für Tags an Klasse

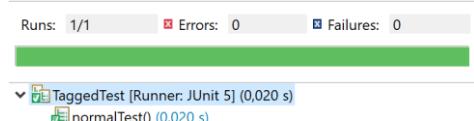
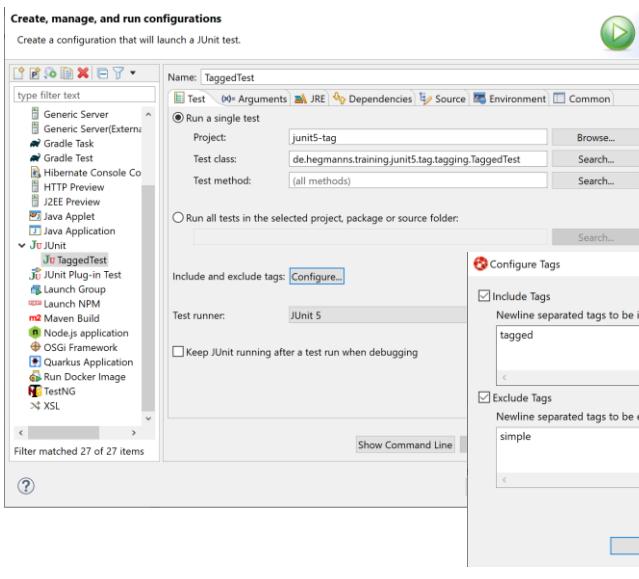
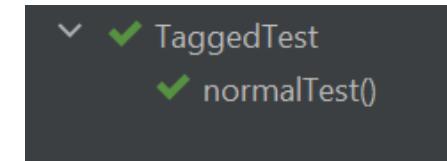
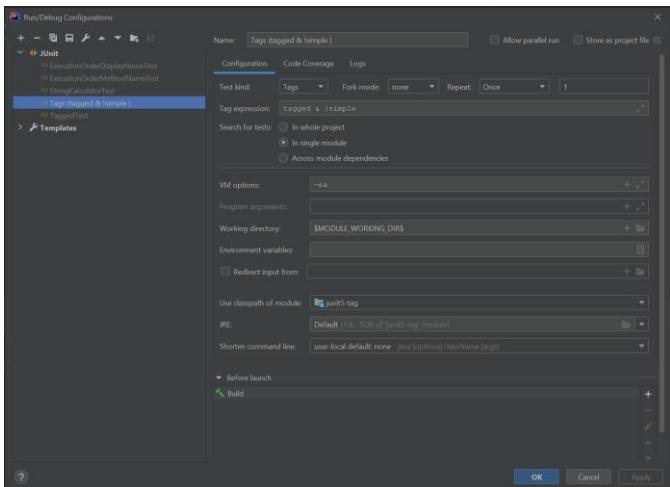
incl	long	tagged	high	simple	high
excl	long			simple	
	X	X		X	X
	X	X	X		X

```
    @Tag("tagged")
    @Tag("high")
    public class TaggedTest {

        @Test
        @Tag("simple")
        public void simpleTest() {
            System.out.println("TaggedTest : simpleTest (tagged, high, simple)");
        }

        @Test
        public void normalTest() {
            System.out.println("TaggedTest : simpleTest (tagged, high)");
        }
    }
```

Tags berücksichtigen in den Entwicklungsumgebungen



Kategorisieren : @Tag Aus/Einschließen in Maven/Gradle

- › @Tag-Annotation
 - › Klasse: Tag gilt für alle Testmethoden
 - › Methode: Tag gilt für diese Methode
- › Tag einschließen
- › Tag ausschließen
 - › Sinnvoll für Tags an Klasse

› Maven

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>${surefire.version}</version>
  <configuration>
    <properties>
      <configurationParameters>
        junit.jupiter.testinstance.lifecycle.default = per_method <!-- this is default and can be omit -->
      </configurationParameters>
      <!--
        junit.jupiter.testinstance.lifecycle.default = per_method
        junit.jupiter.testinstance.lifecycle.default = per_class
      -->
      <groups>high</groups>
      <excludedGroups>simple</excludedGroups>
    </properties>
  </configuration>
</plugin>
```

› Gradle

```
test{
    useJUnitPlatform{ JUnitPlatformOptions it ->
        includeTags 'simple'
        excludeTags 'wrong'
    }
}
```

Auf ein Wort: Tests kategorisieren/ausschließen ?



- › Alle Tests dauern ggf. doch sehr lang'
 - › ... und behindern die Entwicklung
 - › Mit Tags können die relevanten Testcase markiert werden
 - › Auch neue Teammitglieder können so schnell und sicher entwickeln
 - › Prinzipiell müssen alle Tests ausgeführt werden
 - › (auch temporäres) Nicht-Ausführen von Tests kann Fehler verdecken
 - › Ggf. aufwändig in der Korrektur
- › WICHTIG →
- › Tests müssen möglichst häufig ausgeführt werden
 - › Tag-Einsatz während der Entwicklung ist ein probates Mittel
 - › In gewissen Abständen immer alle Tests ausführen (nicht zu lange warten)
 - › Spätestens mit jedem Commit/Push

Alternative Namen: @DisplayName



Alternative Namen sollen die Lesbarkeit verbessern ...

- › Der Methodenname hat gewisse Einschränkungen
 - › Keine Whitespaces
- › Der DisplayName sorgt für eine bessere Darstellung im Testrunner und der Testergebnisdarstellung
 - › Z.B. einigermaßen sinnvolle Sätze
 - › Z.B. Whitespaces

```
@Test
@DisplayName("aReallyBetterNameForThisTest")
public void foo() {
    // don't do such silly things
}

@Test
@DisplayName("sum of 2 and 3 is 5")
public void sumOfTwoAndThreeIsFive() {
    // yeah, that's the reason for giving a proper display name.

    // imagine, if this test goes red. Then the summary says "sum of 2 and 3 is 5 failed". (fine)
}
```

Alternative Namen: @DisplayNameGeneration



Auch in JUNIT 4 gibt es die Möglichkeit der DisplayName-Anpassung

- › Die JUNIT4-Möglichkeiten sind allerdings deutlich umfangsreicher ...



Alternative Namen sollen die Lesbarkeit verbessern ...

- › Annotation @DisplayNameGeneration
 - › Eine Annotation an der Testklasse
- › Stellt quasi ein Regelwerk zur Generierung des DisplayNameen zur Verfügung
- › Das Regelwerk findet sich in einer Klasse
 - › Subklasse von „DisplayNameGenerator“
- › Eigene Regelwerke sind grundsätzlich möglich

```
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)
public class CommonDisplayNameGenerationTest {

    @Test
    public void sum_of_2_and_3_is_5() {
    }
}
```

✓ CommonDisplayNameGenerationTest
✓ sum of 2 and 3 is 5()

Test-Reihenfolge



Reihenfolge in JUnit

- › Reihenfolge von Testklassen und Testmethoden folgt einer Regel
 - › Ist aber grundsätzlich nicht definiert
- › Mitunter stellen die einzelnen Testmethoden Workflow-Schritte dar
- › Es sollte auch keine Reihenfolge innerhalb der Testdurchführung berücksichtigt werden
 - › Es darf auch Keine bestehen
- › Jeder Test muss ...
 - › Unabhängig von anderen Tests sein
 - › Muss auch alleine ausführbar sein



Wenn's doch mal sein muss:

```
@org.junit.jupiter.api.TestMethodOrder(1)
    org.junit.jupiter.api.MethodOrderer.OrderAnnotation.class)
public class YourTest{

    @org.junit.jupiter.api.Test
    @org.junit.jupiter.api.Order(2)
    public void yourTestMethod() {}
```

- › `@TestMethodOrder`-Annotation
 - › An der Klasse zur Kennzeichnung
- › `@Order`-Annotation
 - › Definition der Reihenfolge für jede Testmethode



Gerade in Verbindung mit Remote-Services (Datenbank) und anderen Frameworks (Spring, etc.) können schnell Reihenfolgeeffekte eintreten, daher:

- Testmethoden auch mal alleine ausführen
- Testklassen auch mal alleine ausführen

Einstellungsmöglichkeiten der Ausführungsreihenfolge

Annotation

- › @TestMethodOrderer an Klasse
 - › Als Attribut eines der unteren Klassen
 - › Order eine eigene MethodOrderer-Klasse



Mögliche Attributwerte

Alphanumeric

-Djunit.jupiter.testinstance.lifecycle.default=per_class
bzw. per_method

OrderAnnotation

-junit.jupiter.testinstance.lifecycle.default=per_class
bzw. per_method

Random

Surefire-Plugin: configurationParameter:
junit.jupiter.testinstance.lifecycle.default=per_class

Hinweis:

- › Hintergrund ist die Oberklasse „MethodOrderer“
- › Änderung mit Version 5.7:
 - › „Alphanumeric wird ab Version 5.7 ausgetauscht zu „MethodName“
 - › Neue Orderer ab Version 5.7: DisplayName

Test-Reihenfolge

```
public class WithOrderTest {  
  
    @Test  
    public void firstTestMethod() { }  
  
    @Test  
    public void secondTestMethod() { }  
  
    @Test  
    public void thirdTestMethod() { }  
  
    @Test  
    public void fourthTestMethod() { }  
}
```

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)  
public class WithOrderTest {  
  
    @Test  
    @Order(1)  
    public void firstTestMethod() { }  
  
    @Test  
    @Order(2)  
    public void secondTestMethod() { }  
  
    @Test  
    @Order(3)  
    public void thirdTestMethod() { }  
  
    @Test  
    @Order(4)  
    public void fourthTestMethod() { }  
}
```

- ✓ thirdTestMethod
- ✓ secondTestMethod
- ✓ fourthTestMethod
- ✓ firstTestMethod

- ▼ ✓ WithOrderTest
 - ✓ firstTestMethod()
 - ✓ secondTestMethod()
 - ✓ thirdTestMethod()
 - ✓ fourthTestMethod()

Test-Reihenfolge, ein wenig tiefer geschaut

- › Verantwortlich zur Festlegung der Ausführungsreihenfolge:
 - › Annotation an der Klasse
 - › Eine Klasse vom Typ „org.junit.jupiter.api.MethodOrderer“
 - › Es sind also neben den vorhandenen MethodOrderer auch individuelle Orderer möglich
 - › Ggf. weitere durch den MethodOrderer vorgegebene Annotationen
 - › z.B. OrderAnnotation: @Order

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@API(
    status = Status.EXPERIMENTAL,
    since = "5.4"
)
public @interface TestMethodOrder {
    Class<? extends MethodOrderer> value();
}
```

Test-Lifecycle

Default-Lifecycle

- › **Lifecycle.PER_METHOD**
- › Ausführung jeder Testmethode in eigener Test-Instanz
- › Eine eigene Instanz wird auch für „ausgeschaltete“ Testmethoden erstellt

Alternativ-Lifecycle

- › **Lifecycle.PER_CLASS**
- › Erstellung einer Instanz für alle Test-Methoden
- › @BeforeAll/@AfterAll-Methoden müssen nicht mehr static sein



Einstellmöglichkeiten

Property

-Djunit.jupiter.testinstance.lifecycle.default=per_class
bzw. per_method

Conf-File

(junit-platform.properties)

-junit.jupiter.testinstance.lifecycle.default=per_class
bzw. per_method

Maven

Surefire-Plugin: configurationParameter:
junit.jupiter.testinstance.lifecycle.default=per_class

Gradle

Setzen als System-Properties im test-task

Test-Lifecycle

Default-Lifecycle

- › **Lifecycle.PER_METHOD**
- › Ausführung jeder Testmethode in eigener Test-Instanz
- › Eine eigene Instanz wird auch für „ausgeschaltete“ Testmethoden erstellt

Alternativ-Lifecycle

- › **Lifecycle.PER_CLASS**
- › Erstellung einer Instanz für alle Test-Methoden
- › @BeforeAll/@AfterAll-Methoden müssen nicht mehr static sein



Benefits des „per class“ Lifecycle

- ☞ Simplere Ausführung, ggf. schneller
- ☞ In Verbindung mit Ausführungsreihenfolge könnten leicht Workflow-Schritte getestet werden in einer Testklasse getestet werden
- ☞ Werte in Klasse (Attribute) bleiben unverändert



Übung 08

› Resourcen

de.hegmanns.training.junit5.practice.task08

Die Testklasse „CompleteNumberTest“ enthält auskommentiert die Testmethoden „one()“, „two“, „three()“, „four()“.

Erstellen Sie die Annotation „@OrderByNumber()“, die in einer Testklasse enthaltenen Methodennamen jeweils nach dem ausgeschriebenen Zahlenwert ausgeführt werden.

Die Annotation soll optional noch einen alternativen Sortierer aufnehmen können, der verwendet wird, wenn der Methodename keine ausgeschriebene Zahl ist. Nicht ausgeschriebene numerische Methodenname werden grundsätzlich am Ende ausgeführt, ggf. dann sortiert über den zusätzlichen Sortierer.

Entfernen Sie die Kommentare und stellen Sie die Testklasse so um, dass alle Tests grün werden.
ALLERDINGS ohne die Implementierung in den Testmethoden zu verändern!

DI für Testmethoden

- › Testmethoden / Konstruktoren können allgemein bestimmte Typen als Parameter haben
 - › Diese ersetzen teilweise Rules aus JUnit4
 - › Werden automatisch vom Framework mit Instanzen versorgt

TestReporter

- › Vorgesehen für ggf. nötige Ausgaben
 - › Ausgaben immer über TestReporter-Instanz durchführen

TestInfo

- › Gibt Informationen über Methode, Darstellungsname, Tags, ...
- › Kann beispielsweise mit TestReporter verwendet werden

```
@Test
public void testTheBest(TestReporter testReporter) {
    try {
        Thread.sleep( millis: 1000 );
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    testReporter.publishEntry( value: "that's it" );
}
```

Tests passed: 1 of 1 test - 1 s 60 ms

timestamp = 2020-10-16T22:40:46.439, value = that's it

```
@DisplayName("Java Nashorn script Test")
@Test
void nashornTest(TestInfo testinfo) {
    ScriptEngine engine = new ScriptEngineManager().getEngineByName( shortName: "nashorn" );
    try {
        // remove @Disabled to demonstrating how nashorn script works
        Object result = engine.eval( "(x=2 * 3 == 6)" );
        System.out.println("(2 * 3 == 6) => "+result);
    }

    Bindings bindings = engine.getBindings(ScriptContext.ENGINE_SCOPE);
    bindings.put( name: "systemProperty", System.getProperties() );
    bindings.put( name: "junitDisplayName", testinfo.getDisplayName() );
    bindings.put( name: "junitTags", testinfo.getTags() );
    engine.setBindings(bindings, ScriptContext.ENGINE_SCOPE);

    System.out.println("systemProperty.get('os.arch') => "+engine.eval("script:systemProperty.get('os.arch')"));
    System.out.println("junitDisplayName => "+engine.eval("script:junitDisplayName"));
    System.out.println("junitTags => "+engine.eval("script:junitTags"));
} catch (Exception e) {
    e.printStackTrace();
}
}

Tests passed: 1 of 1 test - 720 ms
C:\Java\jdk8_64\bin\java.exe ...
(x * 3 == 6) => true
systemProperty.get('os.arch') => amd64
junitDisplayName => Java Nashorn script Test
junitTags => []

Process finished with exit code 0
```

Testausführung wiederholen

@RepeatedTest

- › Wiederholung der Testmethode
 - › Jede Ausführung zählt hier quasi als eigene Testmethode
- › Attribute:
 - › Anzahl der Wiederholungen
 - › Angabe eines Display-Namens
 - › Zusätzliche Variablen:
 - › {displayName}
 - › {currentRepetition}
 - › {totalRepetitions}
 - › Kann mit @DisplayName kombiniert werden

```
public class RepeatedExampleWithDisplayNameTest {

    @RepeatedTest(value = 3, name = "Task {currentRepetition} from {totalRepetitions}")
    public void repeatedWithName() {
    }

    @RepeatedTest(value = 3, name = "{displayName}: {currentRepetition}. task from {totalRepetitions}")
    @DisplayName("This is a repeated test")
    public void repeatedWithNameAndDisplayName() {
    }
}
```

- ✓ ✓ RepeatedExampleWithDisplayNameTest
- ✓ ✓ This is a repeated test
 - ✓ This is a repeated test: 1. task from 3
 - ✓ This is a repeated test: 2. task from 3
 - ✓ This is a repeated test: 3. task from 3
- ✓ ✓ repeatedWithName()
 - ✓ Task 1 from 3
 - ✓ Task 2 from 3
 - ✓ Task 3 from 3

Testausführung wiederholen

@RepeatedTest

- › Wiederholung der Testmethode
 - › Jede Ausführung zählt hier quasi als eigene Testmethode
- › Attribute:
 - › Anzahl der Wiederholungen
 - › Angabe eines Display-Namens
 - › Zusätzliche Variablen:
 - › {displayName}
 - › {currentRepetition}
 - › {totalRepetitions}
 - › Kann mit @DisplayName kombiniert werden
 - › Trotz Wiederholung kann in der Testmethode mit jeder Ausführung auch grundsätzlich etwas Anderes passieren
 - › Z.B. durch Auswertung/Verwendung der RepititionInfo-Klasse
 - › Z.B. Initialisierung in BeforeEach-Methode

Benefit

DI für Testmethoden

- › Testmethoden / Konstruktoren können allgemein bestimmte Typen als Parameter haben
 - › Diese ersetzen teilweise Rules aus JUnit4

TestReporter

- 
- › Vorgesehen für ggf. nötige Ausgaben
 - › Ausgaben immer über TestReporter-Instanz durchführen

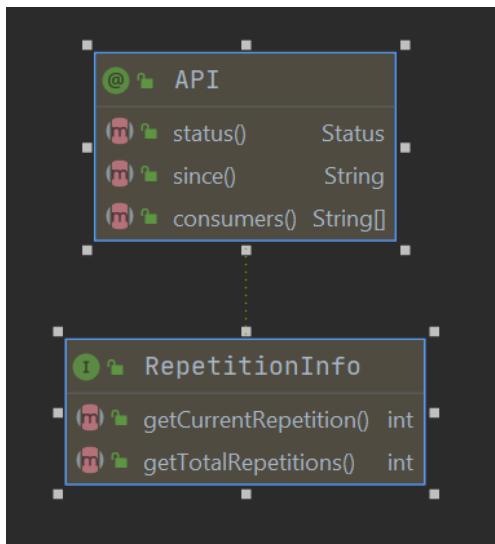
TestInfo

- 
- › Gibt Informationen über Methode, Darstellungsname, Tags, ...
 - › Kann beispielsweise mit TestReporter verwendet werden

RepetitionInfo

- 
- › Für über @RepeatedTest mehrfach Tests
 - › Enthält Information zur aktuellen Ausführung, beispielsweise die lfdn. Nummer

DI für Testmethoden: RepetitionInfo



```
@RepeatedTest(100)
public void checkCounterValue(RepetitionInfo repetitionInfo) {
    MatcherAssert.assertThat(counter.getValue(), Matchers.is(value: repetitionInfo.getCurrentRepetition()-1));
}
```



Übung 09

› Ressourcen

de.hegmanns.training.junit5.practice.task04

de.hegmanns.training.junit5.practice.task09

Im Main-Folder findet sich die Klasse „ de.hegmanns.training.junit5.practice.task04.BoundedCounter“. (Bekannt aus Übung 4)

Auch die zugehörige Testklasse ist bereits bekannt, ist im Test-Folder aber noch mal

eingefügt: „ de.hegmanns.training.junit5.practice.task09.BoundedCounterCalculatesBoundedTest“, allerdings nur die eine Testmethode “boundedCounterDoesntIncrementAboveMaximum()”

Stellen Sie die Testklasse so um, dass mit Hilfe der RepeatetTest-Annotation quasi das gleiche (und ggf. noch ein wenig mehr) getestet werden kann.

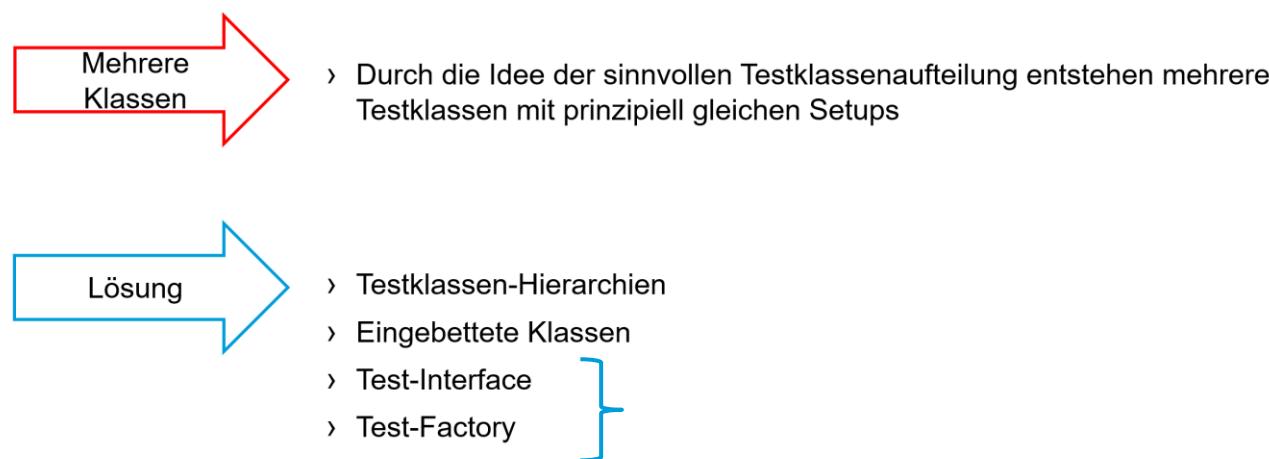
Der Zählerstand von Instanzen dieser Klasse ist jeweils nach oben und unten begrenzt.

Es geht hier ausschließlich um die enthaltene Testmethode und um ein Refactoring hierzu. Weitere Testmethoden/TestCases müssen nicht erstellt werden.

Zur Erinnerung:

Ähnlichkeiten im Setup

- › Ähnliche / Gleiche Setups können in den Before-Lifecycle-Methoden realisiert werden
- › ABER:
 - › Testklasse sollte testgetrieben sein
 - › Use-Case-getrieben
 - › Service-getrieben
 - › Input-getrieben



Test-Interface

- › Viele Vorbereitungen und Ausführungen können in default-Methoden realisiert werden
 - › So können Mechanismen völlig einfach wiederverwendet werden
- › Generische Tests
- › Test-Interface vs. Eingebettete Tests / Test-Hierarchie
 - › Test-Interface: Fokus auf Mechanismen / Technik
 - › Test-Hierarchie / eingebettete Tests: Fokus auf Fachlichkeit

```
public interface BaseShieldDBBuilderTest<T> {  
  
    @Test  
    default void firstShieldPlacedAtBeginningOfShield() {  
        ShieldDBShield<T> anyFirstShield = mock(ShieldDBShield.class);  
        ShieldDBShield<T> anySecondShield = mock(ShieldDBShield.class);  
  
        List<T> build = ShieldDB.<T>builder()  
            .shield(anyFirstShield)  
            .shield(anySecondShield)  
            .build();  
  
        assertThat(build, is(anyFirstShield));  
    }  
  
    @Test  
    default void setNextShieldForNestingShieldsWouldBeProceededByBuilder() {  
        ShieldDBShield<T> anyFirstShield = mock(ShieldDBShield.class);  
    }  
}
```





Übung 10

› Ressourcen

de.hegmanns.training.junit5.practice.task10

Die Testklasse „AggregationTest“ enthält drei nahezu gleich implementierte Testmethoden, die sich nur im verwendeten Typ unterscheiden.

Refactoren Sie diese Klasse in eine Hierarchie mit TestInterface, in dem Sie die bereits vorbereiteten Klassen CommonTestInterface sowie CommonTestFor... verwenden.

Tip: Überlegen Sie sich noch, wie Sie die konkreten Werte in den Test bekommen.

Parallel Testausführung



Grenzen der einfachen Unit-Tests

- › Race-Conditions in der Ausführung
- › Multithread-Probleme
 - › Schreib/Lese-Probleme
 - › DeadLocks



- › Client/Server-Anwendungen/Services
- › Remote-Services
- › Eventbasierte Anwendung

- › Thread-Problematik nicht immer sofort erkennbar
- › Thread-Problematik durch allgemeine Einzeltests selten lokalisierbar
- › Thread-Problematik häufig durch Penetrationstests erkennbar



Bei Anwendungen mit Multithreading können unterschiedliche zusätzliche Probleme auftreten

- Multithread-Einsatz innerhalb der Testmethode möglich
 - Der Testfokus wird aber verdeckt
- Unterstützung durch Test-Framework

Parallel Ausführung

```
@Execution(ExecutionMode.CONCURRENT)
public class ParallelTest {

    @Test
    void test01() {
        System.out.println(Thread.currentThread().getName() + " test01");
        try {
            Thread.sleep( 500 );
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Test
    void test02() {
        System.out.println(Thread.currentThread().getName() + " test02");
        try {
            Thread.sleep( 500 );
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Konfiguration in „junit-platform.properties“

```
junit.jupiter.execution.parallel.enabled = true
#junit.jupiter.testinstance.lifecycle.default = per_method
```

Jupiter-Assertions: Parameterreihenfolge



Aufpassen bei der Parameter-Reihenfolge

- › Die Methoden von Assertions verhindern nicht ein Vertauschen von Sollwert und Ist-Wert
- › Für fehlgeschlagene Tests können verwirrende / falsche Meldungen entstehen

```
public class SearchingForWhatIsExpectedAndWhatIsGivenTest {  
  
    private final Integer expectedResult = 5; // try with 4 for getting the tests green  
  
    @Test  
    public void useRightOrder() {  
        int result = 2+2;  
  
        Assertions.assertEquals(expectedResult, result, message: "problem adding two numbers");  
    }  
  
    @Test  
    public void useFalseOrder() {  
        int result = 2+2;  
  
        Assertions.assertEquals(result, expectedResult, message: "problem adding two numbers");  
    }  
}
```



```
org.opentest4j.AssertionFailedError: problem adding two numbers ==>  
Expected :5  
Actual   :4
```

```
org.opentest4j.AssertionFailedError: problem adding two numbers ==>  
Expected :4  
Actual   :5
```

Assertions: Object als Parametertyp



Aufpassen bei Parameter-Typen

- › Es gibt umfangreich überladene Methoden mit paarweise gleichen Typen
- › Aber es gibt auch als Parameter Object
(hier darf Soll und Ist anderer Typ sein)
- › Je nach Konstellation/Klasse von Soll und Ist können sich schwer zu erkennende Fehlerkonstellationen ergeben

```
public class SecurityServiceTest {  
    private final SecurityService securityServiceUnderTest = new SecurityService();  
  
    @Test  
    public void securityServiceReturnsTheRightSecurityForGoodCustomers() {  
        Customer anyGoodCustomer = new Customer( customerId: 12345, goodCustomer: true);  
  
        SecurityNumber security = securityServiceUnderTest.getSuitableSecurity(anyGoodCustomer);  
  
        Assertions.assertEquals( expected: "710000", security);  
    }  
}
```



```
org.opentest4j.AssertionFailedError: expected: java.lang.String@7bc7ad2e<710000> but was: de.hegmanns.training.junit5.example001.demo001  
.SecurityNumber@42950643<710000>  
Expected :710000  
Actual   :710000  
*Click to see differences*
```

Tipp: Verwendung alternativer Vergleichs-Framework



Vergleichsframework Hamcrest

Vergleichsframework AssertJ / Truth

- › Hamcrest/AssertJ ist beim Soll und Ist **typischer**
 - › Unterschiedliche Typen erzeugen einen Compile-Fehler
- › Hamcrest/AssertJ ist beim Soll und Ist in der Reihenfolge definiert/**eindeutig**
 - › So sind verwirrende/falsche Meldungen bei roten Tests vermeidbar
- › Hamcrest/AssertJ entspricht einer natürlichen Leseart



Hamcrest / AssertJ:

- Mit Hamcrest/AssertJ wird Rapid-Development unterstützt
- Hamcrest/AssertJ-Comparatoren sind erweiterbar

Hamcrest und AssertJ

Hamcrest

- › Zwei Kernklassen
 - › MatcherAssert
 - › Matchers
- › Keine Fluent-API
- › Kombination eher leicht

```
private int givenInt = 2;
private int expectedInt = 2;

@Test
public void jupiterSimpleAssert() {
    org.junit.jupiter.api.Assertions.assertEquals(expectedInt, givenInt);
}

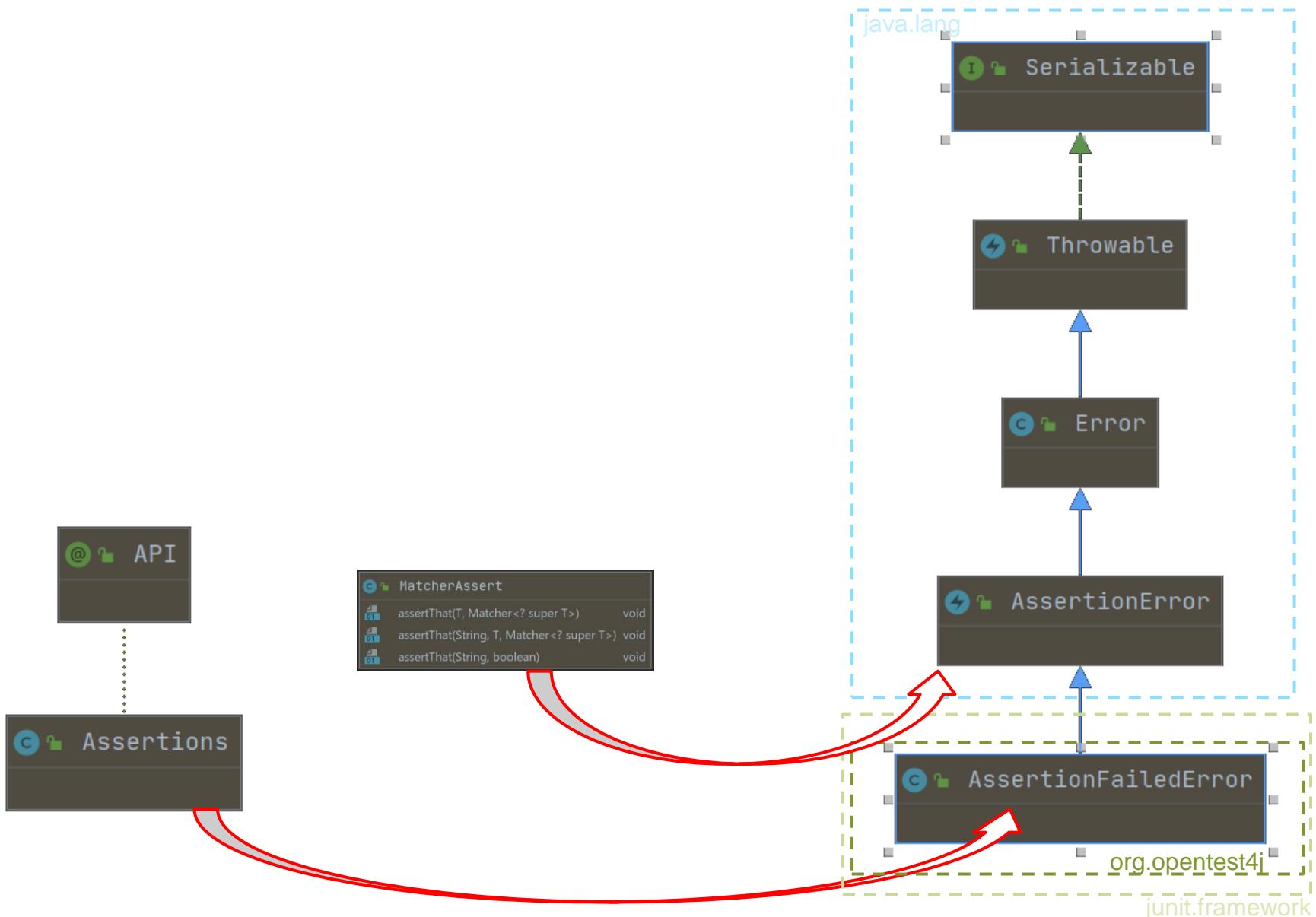
@Test
public void hamcrestSimpleAssert() {
    org.hamcrest.MatcherAssert.assertThat(actual: 2, org.hamcrest.Matchers.is(expectedInt));
}

@Test
public void assertJSimpleAssert() {
    org.assertj.core.api.Assertions.assertThat(givenInt).isEqualTo(2);
}
```

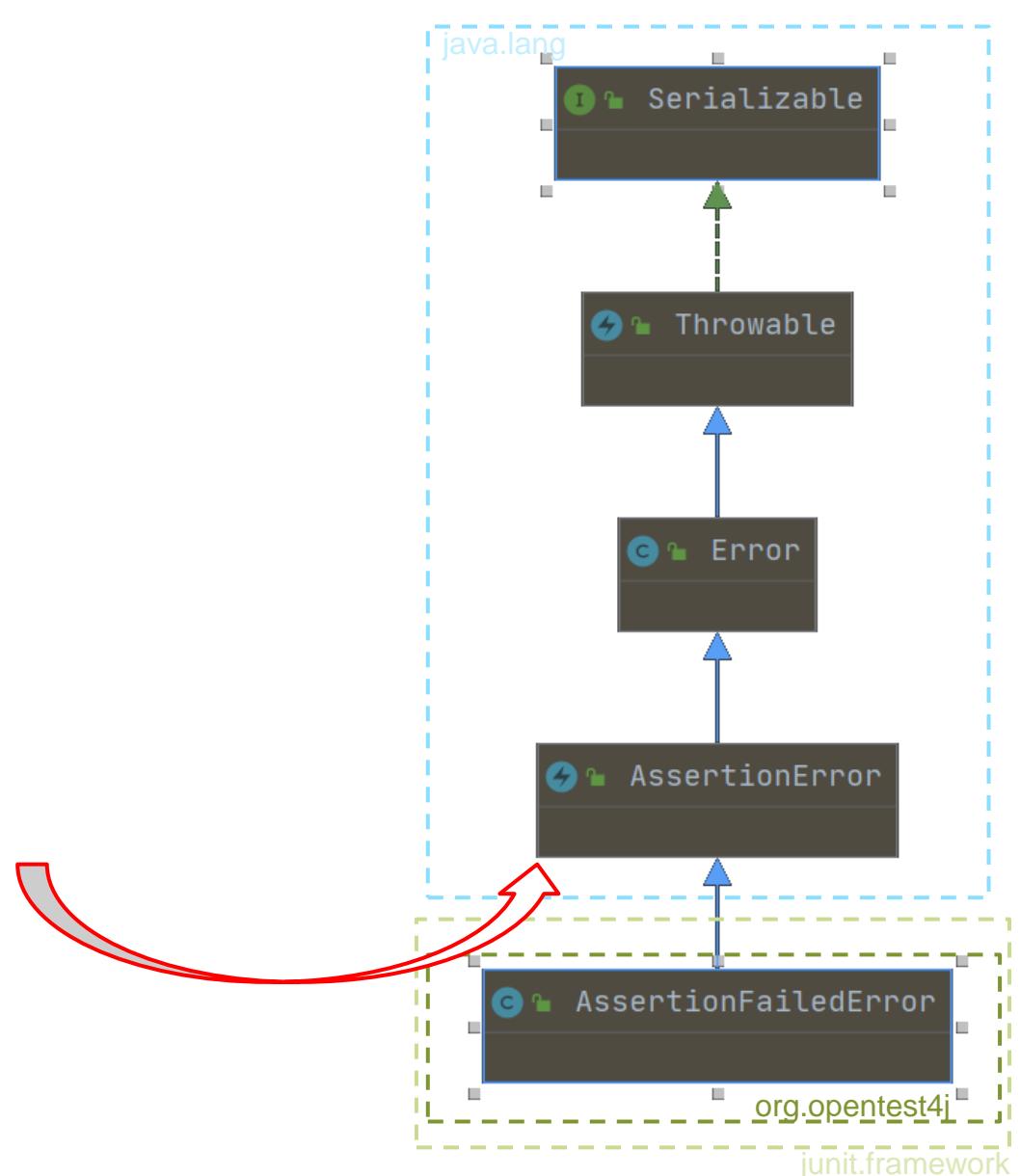
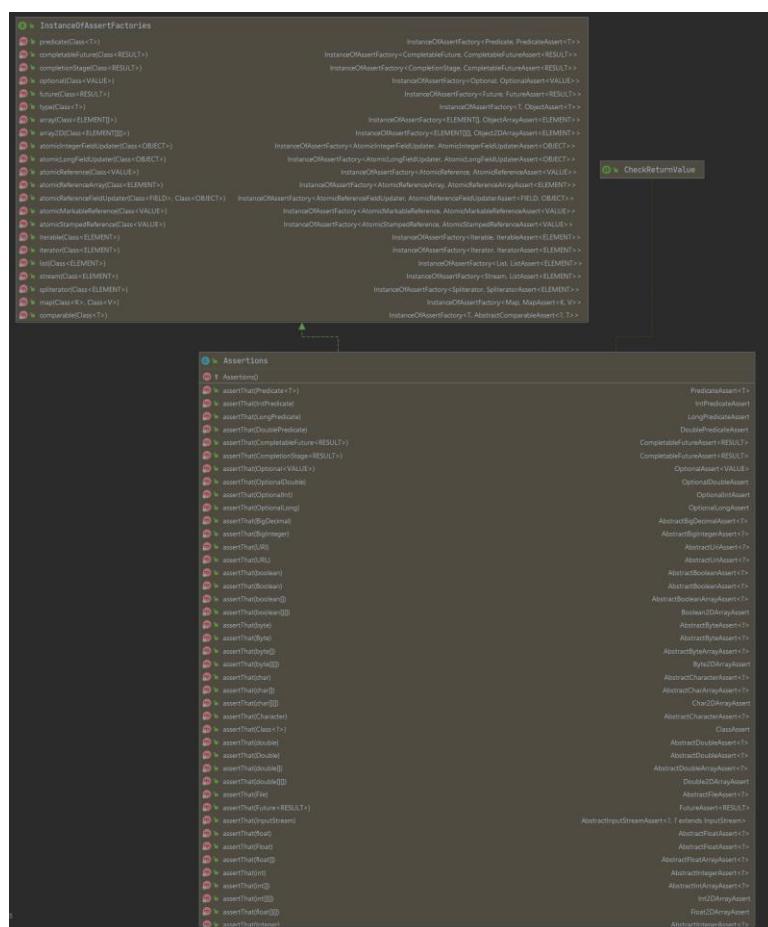
AssertJ

- › Eine Kernklassen
 - › Assertions
 - › Entspricht eher einer Fluent-API
- › In Implementation noch einen Komparator
- › Beliebige Kombination (z.B. mit not) eher schwierig

Vergleichsframework Hamcrest, kleiner Nachschlag



Vergleichsframework AssertJ, kleiner Nachschlag



Hamcrest: Wichtige Klassen

Paket

org.hamcrest

Assert-Klasse

org.hamcrest.MatcherAssert

Facade

org.hamcrest.Matchers

Matchers

In weiteren Packages, je nach Typ.

Z.B. org.hamcrest.core.IsSame

z.B. org.hamcrest.collection.IsMapWithSize

Hamcrest: Vorgehensweise

Paket

org.hamcrest

Assert-Klasse

- › MatcherAssert-Klasse
 - › Stellt Einstiegspunkt der Zusicherungsprüfung dar
- › Parameter:
 - › Zu vergleichende Instanz (given)
 - › Matcher (Zuständig für Vergleiche)
 - › Zusätzliche Textinformation im Fehlerfall
 - › Kann mit Hamcrest **häufig entfallen** bzw. sehr klein gehalten werden
 - › Enthält bei Hamcrest häufig fachliche Erläuterung

```
MatcherAssert.assertThat(yourInstance, <Matcher>)
MatcherAssert.assertThat(errorText, yourInstance, <Matcher>)
MatcherAssert.assertThat(errorText, boolean)
```

```
MatcherAssert.assertThat(anyCustomer, Matchers.hasProperty( propertyName: "lastName", Matchers.is( value: "Hegi")));
```

```
java.lang.AssertionError:
Expected: hasProperty("lastName", is "Hegi")
      but: property 'lastName' was "Hegmanns"
```

Hamcrest: Vorgehensweise

Paket

org.hamcrest

Assert-Klasse

- › MatcherAssert-Klasse
 - › Stellt Einstiegspunkt der Zusicherungsprüfung dar
- › Parameter:
 - › Zu vergleichende Instanz (given)
 - › Matcher (Zuständig für Vergleiche)
 - › Zusätzliche Textinformation im Fehlerfall
 - › Kann mit Hamcrest häufig entfallen bzw. sehr klein gehalten werden
 - › Wenn, dann bei Hamcrest häufig **fachliche Erläuterung**

```
MatcherAssert.assertThat(yourInstance, <Matcher>)
MatcherAssert.assertThat(errorText, yourInstance, <Matcher>)
MatcherAssert.assertThat(errorText, boolean)
```

```
MatcherAssert.assertThat(reason: "customers last name", anyCustomer, Matchers.hasProperty(propertyName: "lastName", Matchers.is(value: "Hegi")));
```

```
java.lang.AssertionError: customers last name
Expected: hasProperty("lastName", is "Hegi")
but:   property 'lastName' was "Hegmanns"
```

Hamcrest: Vorgehensweise

Paket

`org.hamcrest`

Matcher

- › Matcher
 - › Typgebundene Zusicherungschecks
 - › Häufig erreichbar über Fassadenklasse:
 - › **Matchers** (CoreMatchers)
 - › Teilweise fordern einige Matcher wiederum Matcher
(z.B. Matchers.is.Matchers.lowerThan(5)))
 - › Teilweise wird ein direkter Vergleich durchgeführt
(z.B. Matchers.is(3))

Human readable/sugar

- › In Verbindung mit „is“, „not“ im Ansatz BDD möglich
 - › Matchers.is(<Matcher>)
 - › Matchers.not(<Matcher>)

Hamcrest: Fassadenklassen

Die Matchers-Klasse gibt die eigentlichen Matcher-Instanzen zurück, die (je nach konkretem Typ) in unterschiedlichen Klassen liegen

Matchers

```
...  
Matcher<CharSequence> hasLength(int length)  
Matcher<String> stringContainsInOrder(String... substring)
```

CharSequenceLength

```
...  
Matcher<CharSequence> hasLength(int length)  
...
```

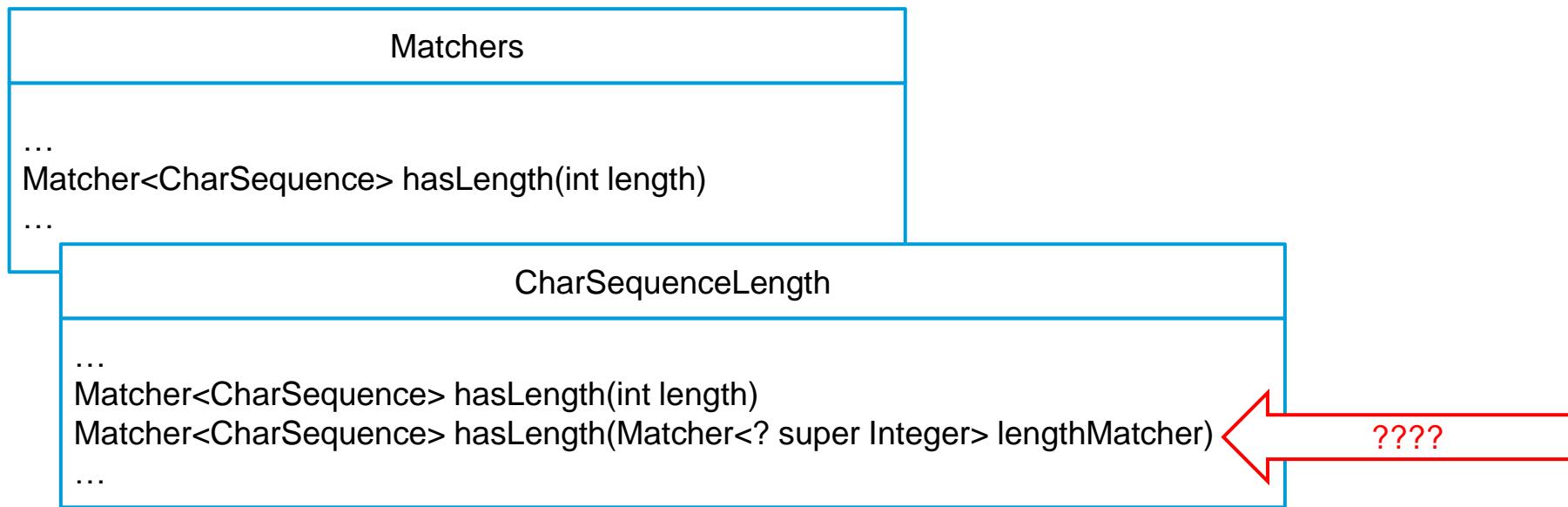
StringContainsInOrder

```
...  
Matcher<String> stringContainsInOrder(String... substring)  
...
```

Hamcrest: Hinter der Facade steckt auch noch etwas ...

Die Matchers-Klasse gibt die eigentlichen Matcher-Instanzen zurück, die (je nach konkretem Typ) in unterschiedlichen Klassen liegen.

Die Matchers-Klasse hat nicht zu jedem Matcher eine Fassaden-Methode.



Prinzipiell können auch die Matcher-Klasse direkt verwendet werden:

```
MatcherAssert.assertThat(aSimpleString, Matchers.hasLength(9));  
  
MatcherAssert.assertThat(aSimpleString, CharSequenceLength.hasLength(Matchers.lessThan( value: 20)));  
MatcherAssert.assertThat(aSimpleString, CharSequenceLength.hasLength(Matchers.greaterThan( value: 5)));
```

Hamcrest: Verwendung mit/ohne static-import

```
import static org.hamcrest.MatcherAssert.*;
import static org.hamcrest.Matchers.*;

import org.junit.jupiter.api.Test;
import java.math.BigDecimal;
import java.math.BigInteger;
import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class HamcrestExamplesWithStaticImportTest {

    @Test
    public void forSimpleValue() {
        int aSimpleInt = 6;

        assertThat(aSimpleInt, is(value(6)));
        assertThat(aSimpleInt, lessThan(value(10)));
        assertThat(aSimpleInt, lessThanOrEqualTo(value(10)));
        assertThat(aSimpleInt, greaterThan(value(0)));
        assertThat(aSimpleInt, greaterThanOrEqualTo(value(0)));
    }
}
```

- › MatcherAssert und Matchers mit static import
 - › Kürzerer Schreibweise
 - › Besserere Lesbarkeit
- › Vorsicht bei Doppeldeutigkeiten mit anderen Frameworks/Bibliotheken

- › MatcherAssert und Matchers ohne static import
 - › Individuellere Imports
 - › Granulare Definition der Nutzung

```
import org.hamcrest.MatcherAssert;
import org.hamcrest.Matchers;
import org.junit.jupiter.api.Test;

import java.math.BigDecimal;
import java.math.BigInteger;
import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class HamcrestExamplesTest {

    @Test
    public void forSimpleValue() {
        int aSimpleInt = 6;

        MatcherAssert.assertThat(aSimpleInt, Matchers.is(value(6)));
        MatcherAssert.assertThat(aSimpleInt, Matchers.lessThan(value(10)));
        MatcherAssert.assertThat(aSimpleInt, Matchers.lessThanOrEqualTo(value(10)));
        MatcherAssert.assertThat(aSimpleInt, Matchers.greaterThan(value(0)));
        MatcherAssert.assertThat(aSimpleInt, Matchers.greaterThanOrEqualTo(value(0)));
    }
}
```

AssertJ: mit Fluent-API noch ein Stückchen einfacher

Paket

org.assertj.core.api

Assertions

- › Fassade für das zu checkende Objekt
- › Aufgebaut nach Builder-Pattern

Dann Zusicherung

- › Typabhängiger Vergleich



Übung 11

› Resourcen

de.hegmanns.training.junit5.practice11

Im Testbereich liegen die Testklassen „CounterJupiterTest“, „CounterHamcrestTest“, „CounterAssertJTest“.

Alle drei Testklassen haben momentan die gleichen Testmethoden mit gleichem Inhalt unter Verwendung der Jupiter-Assertions.

Schreiben Sie die Tests in ...HamcrestTest so um, dass Hamcrest-Zusicherungen verwendet werden.

Schreiben Sie die Tests in ...AssertJTest so um, dass AssertJ-Zusicherungen verwendet werden.

Hamcrest-Komparatoren/Matcher

Komplexe Objekte

- › Attributweise Checks nötig
 - › über equals-Methode
 - › Nicht immer deckt die equals-Methode einen attribut-weise Check ab
 - › Equals nützt auch nicht immer ...
 - › Assertions Attribut-Weise
 - › Mühsam
 - › Erzeugt ggf. Fehlermeldungen unterschiedlicher Qualität
 - › Sollte mit assertAll(..) kombiniert werden
 - › Eigene Komparatoren

Komparator

- › Vergleicht zwei Objekte individuell miteinander
 - › Abhängig von der Fachlichkeit
 - › Z.b. alle Attribute
 - › Nur relevante Attribute
 - › Muss natürlich nicht nur equals sein
 - › Definierte Fehlermeldungen aus dem Komparator

Zwei Varianten für Hamcrest-Comparatoren/Matcher

Attribut-Locator

- › Gibt den gewünschten Attributwert zurück
- › Kann dann mit vorhandenen Hamcrest-Matchern verglichen werden
- › Eignet sich eher bei Vergleichen mit genau einem Attributwert
- › Eigene Beschreibungstexte möglich
 - › Information zum Attribut
 - › Information zum Fehler
- › Erstellt wird zwar ein Comparator, ist aber eigentlich ein Locator für den Wert

Attributwert ermitteln

- › Eigene Klasse
- › Führt den gewünschten Vergleich durch
 - › Kann auch für mehrere (verschiedene, unabhängige) Vergleiche zuständig sein
- › Vorteilhaft, wenn gleich mehrere Attribute miteinander verglichen werden müssen
- › Gibt auch Beschreibungstexte zurück

Zwei Varianten für Hamcrest-Comparatoren/Matcher

Attribut-Locator

- › Gibt den gewünschten Attributwert zurück
- › Kann dann mit vorhandenen Hamcrest-Matchern verglichen werden
- › Eignet sich eher bei Vergleichen mit genau einem Attributwert
- › Eigene Beschreibungstexte möglich
 - › Information zum Attribut
 - › Information zum Fehler
- › Erstellt wird zwar ein Comparator, ist aber eigentlich ein Locator für den Wert

- › Idealerweise eine Instanz der Klasse „org.hamcrest.FeatureMatcher“ zurück geben (häufig anonym)
 - › 1. Generic-Typ: die zu vergleichende Klasse
 - › 2. Generic-Typ: Typ des zu vergleichenden Attributes
 - › Text für Fehler
 - › Text zur Wert-Beschreibung
- › Überschreiben der Methode „featureValueOf(...)“
 - › Die Methode gibt den zu vergleichen Attributwert zurück

Zwei Varianten für Hamcrest-Comparatoren/Matcher

Attributwert ermitteln

- 
- › Eigene Klasse
 - › Führt den gewünschten Vergleich durch
 - › Kann auch für mehrere (verschiedene, unabhängige) Vergleiche zuständig sein
 - › Vorteilhaft, wenn gleich mehrere Attribute miteinander verglichen werden müssen
 - › Gibt auch Beschreibungstexte zurück
 - › Idealerweise eine Subklasseder Klasse „org.hamcrest.TypeSafeMatcher“ erstellen
 - › Static-Methoden zur Rückgabe dieser Klasse für die Vergleiche
 - › „matchesSafely“ überschreiben
 - › Könnte auch eine Facade für mehrere Vergleiche
 - › „describeTo“ überschreiben
 - › Beschreibt das given-Objekt
 - › „describeMismatchSafely“ überschreiben
 - › Beschreibt die Erwartung im Fehlerfall



Übung 12

› Resourcen

de.hegmanns.training.junit5.practice12

Wir wollen weiterhin die Funktion der Counter/BoundedCounter-Klasse testen.

Im Package befindet sich die Testklasse CounterTest, die auf sehr einfache Weise die values zweier Counter-Instanzen vergleicht.

Schreiben Sie hierfür einen Hamcrest-Matcher / AssertJ-Komparator, der im Fehlerfall auch eine sinnvolle Fehlermeldung ausgibt.

Verwendet werden soll der Matcher gemäßt:

MatcherAssert.assertThat(firstCounter, lessThan(secondCounter))

ODER:

in AssertJ:

CounterAssertion.assertThat(firstCounter).lessThan(secondCounter)

Tipps für eigene Komparatoren

- › Eigene Komparatoren vereinfachen Tests
 - › Kapseln für neue Teammitglieder komplexe Fachlichkeit
 - › Volle Verantwortung auf Vergleich und Fehlertext
- › Erstellen Sie für komplexe Vergleiche eigene Komparatoren
- › Erstellen Sie einen Locator/Facade für verschiedene Komparatoren einer Fachlichkeit
 - › Z.B. Objekttyp
 - › Z.B. Use-Case-Typ
 - › Z.B. Projekt/Modul

Aus 1 mach n ...

- › Ähnliche/Gleiche Testausführungen mit verschiedenen Eingangs- und Ausgangssituationen
- › Ähnliche/Gleiche Testausführungen mit generierbare Algorithmen

- › Tests mit Parameter und verschiedenen Quellen für die Parameterwerte
 - › Testmethode wird entsprechend oft ausgeführt
- › Generierung von neuen Testmethode (quasi eine Testfactory)
 - › Es entstehend aus einer Testmethode (der Testfactory) entsprechend viele neue Tests

Banale Lösung

- › Innerhalb der Testmethode mehrere Asserts
 - › Ggf. mit assertAll verknüpfen

ABER



- › Jedes Executable benötigt eigene Testvorbereitung
- › Jedes Executable ist quasi sein eigener Test
- › Keine Darstellung im Testrunner

Parametrierte Tests

- › Eine Testmethode mit mehreren speziellen Parameter
 - › Typ ist prinzipiell beliebig
- › Die Parameter können über verschiedene Quelle belegt werden
 - › Möglichkeit angepasster Tests

Prinzipiell könnte man eine entsprechende Absicherung auch durch mehrere Asserts erwirken



Vorteile parametrierter Tests

- 👉 Einfachere Review-Möglichkeiten
- 👉 Darstellung im TestRunner

Input-Quellen für parametrierte Tests

Explizite Werteliste

`@ValueSource(ints={...})`

Liste als csv

`@CsvSource({„1,2“, „5, 10“})`
`@CsvFileSource`

Verweis auf Methode

`@MethodSource(„method“)`

Elemente eines Enum

`@EnumSource(...)`

Elemente werden von einem
Provider geliefert

`@ArgumentSource(ArgumentsProvider)`

Sonderfälle für Werteliste (@ValueSource)

@NullSource

null als Einzelement

@EmptySource

Für Collection-Typen eine leere Collection

@NullAndEmptySource

@MethodSource(„method“)

Sonderfälle für Enum

Typisierung in Methodensignatur

null als Einzelement

Verwendung der String-Repräsentation

Für Collection-Typen eine leere Collection

Ausschluss von Werten

@MethodSource(mode=EXCLUDE, names=..)

Match mit RegEx

@MethodSource(mode=MATCH_ALL, names=..)

Zusammenführen von Parametern: Aggregator

Werteliste

```
@ParameterizedTest  
@CsvSource  
void ihrTest(ArgumentsAccessor arguments){}
```

› ArgumentAccessor

getString(index)
get(index, Class)

```
@ParameterizedTest  
@CsvSource({ "A, XETRA, LIMIT", "A, XETRA, AUCTION_ONLY" })  
public void containsInPossibleSupplements(ArgumentsAccessor argumentsAccessor) {  
    String xetraType = argumentsAccessor.getString( 0 );  
    String exchange = argumentsAccessor.getString( 1 );  
    Supplement expectedSupplement = Supplement.valueOf( argumentsAccessor.getString( 2 ) );
```

Zusammenführen von Parametern: ausgelagerter Aggregator für schwere Fälle ...

Werteliste

```
@ParameterizedTest  
@CsvSource  
void ihrTest(@AggregateWith(YourAggregator.class) Typ param){}
```

YourAggregator
implements
ArgumentAggregator

Typ aggregateArguments(
 ArgumentsAccessor accessor,
 ParameterContext parameterContext)
throws ArgumentAggregationException

```
@ParameterizedTest  
@CsvSource({{"A, XETRA, LIMIT", "A, XETRA, AUCTION_ONLY"}})  
public void containsInPossibleSupplementsWithAggregator(@AggregateWith(OrderAggregator.class) Map<Order, Supplement> orderAndExpectedSupplement) {  
    Map.Entry<Order, Supplement> entry = orderAndExpectedSupplement.entrySet().iterator().next();  
    Assertions.assertThat(locator.getPossibleSupplements(entry.getKey())).containsAnyOf(entry.getValue());  
}
```

```
static class OrderAggregator implements ArgumentsAggregator {  
  
    @Override  
    public Map<Order, Supplement> aggregateArguments(ArgumentsAccessor argumentsAccessor, ParameterContext parameterContext) throws ArgumentsAggregationException {  
        String xetraType = argumentsAccessor.getString(0);  
        String exchange = argumentsAccessor.getString(1);  
        Supplement expectedSupplement = Supplement.valueOf(argumentsAccessor.getString(2));  
  
        Exchange xetra = new Exchange();  
        xetra.setId("123");  
        xetra.setExchangeName(exchange);  
    }  
}
```

Konvertierung von Parametern

› Primitive Konvertierung

Einfache Wertekonvertierung über primitive Datentypen.
(also int-Liste in long, float, etc)

› Implizite Konvertierung

Bei Csv-Inhalt implizite Konvertierung

boolean / Boolean	java.time.LocalDate
byte / Byte	java.time.LocalTime
char / Character	java.time.MonthDay
short / Short	java.time.OffsetDateTime
int / Integer	java.time.OffsetTime
long / Long	java.time.Period
float / Float	java.time.YearMonth
double / Double	java.time.Year
Enum subclass	java.time.ZonedDateTime
java.io.File	java.time.ZoneId
java.lang.Class	java.time.ZoneOffset
java.lang.Class	java.util.Currency
java.math.BigDecimal	java.util.Locale
java.math.BigInteger	java.util.UUID
java.net.URI	
java.net.URL	
java.nio.charset.Charset	
java.nio.file.Path	
java.time.Duration	
java.time.Instant	

Konvertierung von Parametern

› Primitive Konvertierung

Einfache Wertekonvertierung über primitive Datentypen.
(also int-Liste in long, float, etc)

› Implizite Konvertierung

Einfache Wertekonvertierung über primitive Datentypen.
(also int-Liste in long, float, etc)

› Explizite Konvertierung

Verwendung eines Converter. (damit geht quasi alles)

```
void yourTest(@ConvertWith(YourConverter.class) Typ param){}
```

```
class YourConverter extends SimpleArgumentConverter{}
```

```
class YourConverter extends TypedArgumentConverter<A, B>{}
```

› TypesArgumentConverter

 › Für eine simple Konvertierung

› SimpleArgumentConverter

 › Für komplexere Situationen

› Zeit mit Pattern

```
@JavaTimeConversionPattern(<Pattern>)
```

TestFactory

- › Aus einer Methode werden – beliebig viele – individuelle Tests erstellt
 - › Die Tests sind im Testrunner erkennbar
- › Wichtige Elemente
 - › Annotation: @TestFactory
 - › Typ der Methoden: Array, Collection, Stream vom Typ „DynamicTest“



Übung 13

› Resourcen

de.hegmanns.training.junit5.practice01

Im Package „de.hegmanns.training.junit5.practice01“ befindet sich die Testklasse „EvenDividableManyTest“ mit vielen gleichartigen Tests.

Refactoren Sie die Klasse, in dem Sie parametrierte Tests verwenden.



Übung 14

› Resourcen

de.hegmanns.training.junit5.practice006

Wir widmen uns noch mal den Testmöglichkeiten der Cube-Klasse zu.

a)

Wir wollen annehmen, dass ein Cube als fair gilt, wenn von 120 durchgeföhrten Versuchen mindestens 12 mal jede Zahl vorhanden ist.

Testen Sie das Verhalten.

Erstellen Sie hierzu die Klasse FairCubeTest.

b)

Testen Sie den Cube sowohl im Fair als auch im unfair-Modus, in dem Sie jeweils 120 Versuche durchführen und entweder kontrollieren, dass ein Fairer Cube auch wirklich die Minimalverteilung für die Zahlen aufweist und ein unfairener Cube immer die Zahl 6 würfelt.

Erstellen Sie die hierzu nötigen Tests im die Klasse ModeCubeTest.

Dateien/Pfade mit automatischem Aufräumen: @TempDir

- › Verwaltung eines Pfad/einer Datei durch JUnit5
 - › Als Feld möglich (auch statisch)
 - › Als Parameter der Testmethode möglich
- › Pfad wird nach Scope-Ende wieder entfernt
 - › Ggf. vorhandene Dateien werden iterativ entfernt

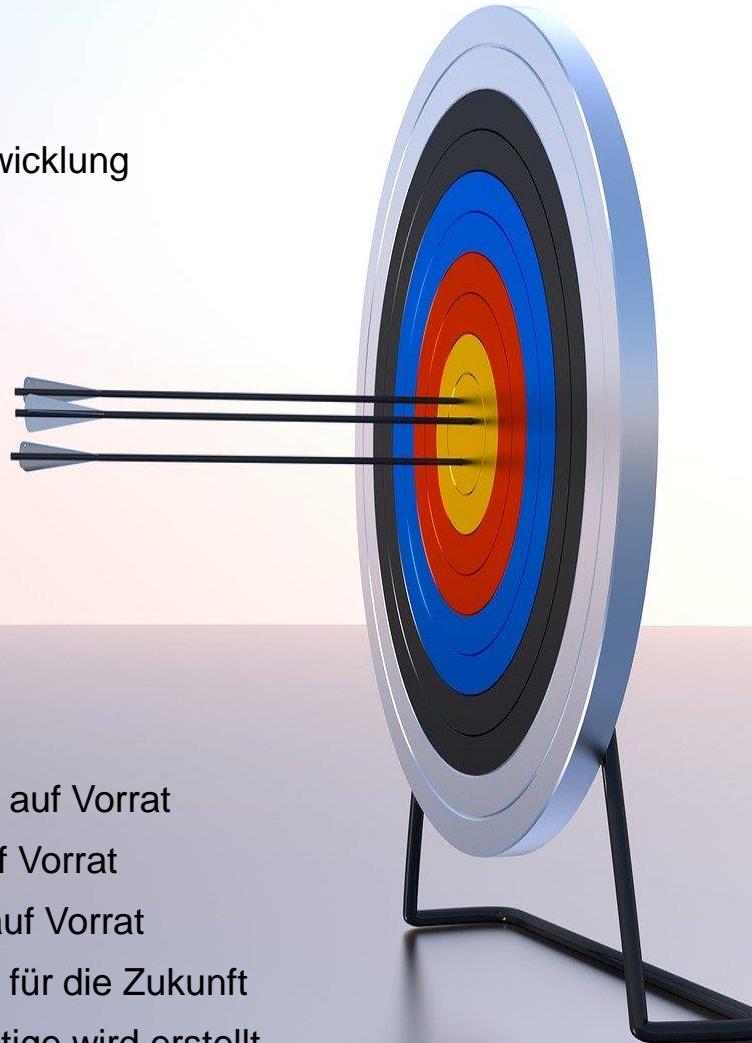
› Falls Aufräumen nicht möglich

› Führt aber nicht zu einem roten Test

```
SCHWERWIEGEND: Caught exception while closing extension context: org.junit.jupiter.engine.descriptor.ClassExtensionContext@5df3d3d5
java.io.IOException: Failed to delete temp directory C:\Users\hegma\AppData\Local\Temp\junit198242535507219901. The following paths could not be deleted (see suppressed exceptions for details):
<23 internal calls>
  at java.base/java.util.concurrent.RecursiveAction.exec(RecursiveAction.java:189)
  at java.base/java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:290)
  at java.base/java.util.concurrent.ForkJoinPool$WorkQueue topLevelExec(ForkJoinPool.java:1020)
  at java.base/java.util.concurrent.ForkJoinPool.scan(ForkJoinPool.java:1656)
  at java.base/java.util.concurrent.ForkJoinPool.runWorker(ForkJoinPool.java:1594)
  at java.base/java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:177)
Suppressed: java.nio.file.FileSystemException: C:\Users\hegma\AppData\Local\Temp\junit198242535507219901: Der Prozess kann nicht auf die Datei zugreifen, da sie von einem anderen Prozess verwendet wird.
```

TDD

- › Test Driven Development
- › Eine agile Methode der Softwareentwicklung
 - › Für schnelle Ergebnisse
 - › Für gute Ergebnisse
 - › Einfach
 - › Klar
 - › Für leichte Ergebnisse
 - › Wenig komplex
 - › Fehlerfrei
 - › Keine Lösung auf Vorrat
 - › Kein Code auf Vorrat
 - › Kein Design auf Vorrat
 - › Keine Lösung für die Zukunft
 - › Einzig das Nötige wird erstellt



TDD Grundidee

- › Erzeugung in kleinen Schritten in definierter Reihenfolge
 - › Test erstellen und ausführen
 - › Bei rotem Test: Test korrigieren nach grün
- › TDD erzeugt konzeptuell vollständig getesteten Code
- › TDD erzeugt konzeptuell ein gutes Design
- › TDD erzeugt keinen unnötigen Code
- › TDD kann bei Neuimplementierungen angewendet werden
- › TDD kann auch bei Featureerweiterungen angewendet werden
- › TDD kann bei Bugs/Fehlern angewendet werden
- › TDD erfordert Disziplin
- › TDD kann nachträgliche Fehlersuche vermeiden
- › TDD hilft bei der Fehlersuche

TDD: Vorgehensweise zur Featureerweiterung/Neuentwicklung

- › Zerlegung in kleine Schritte
- › Jeder Schritt soll ein Verhaltensmerkmal beschreiben
 - › Die Schritte müssen aufeinander aufbauen
 - › Die Schritte sollen sich nicht widersprechen

TDD: Vorgehensweise zur Fehlerbehebung

- › Vorhandene Tests ausführen
- › Tests zur Fehlernachstellung erstellen und ausführen
 - › Der Fehler ist nachgestellt, wenn ein roter Test erstellt ist
- › Bei komplexeren Fehlern Strategie/Lösungskonzept erstellen und in kleine Schritte zerlegen
- › Fehler korrigieren, so dass der nachgestellte Fehlertest grün wird
 - › Bei kleinen Schritten zusätzlich nach TDD vorgehen
(also jeder Schritt erzeugt zunächst einen roten Test)
- › Je nach Komplexität auch auf andere Tests achten
 - › Ggf. erfordert dies eine Anpassung des Lösungskonzepts
- › Wenn Fehler nachweislich gelöst, alle Tests ausführen
 - › Bei nun roten Tests wieder Strategie/Lösungskonzept erstellen

TDD: Ein erstes Beispiel

Schritt 1:

Die Methode „public static int calculate(String string)“ der Klasse „StringCalculator“ gibt für einen null-String den Wert 0 zurück.

So schwer es fällt ...

- › NICHT schon mal die Klasse „StringCalculator“ erstellen.
- › NICHT schon mal die Methode calculate erstellen
- › NICHT vordenken, volle Konzentration auf den Schritt

Test schreiben

- › Der Test/die Tests testen genau (und nur das), was im jeweiligen Schritt beschrieben ist
- › Häufig kompiliert's dann im ersten Schritt nicht
 - › Perfekt, falscher geht's nicht ;)

Test grün bekommen

- › Features der Entwicklungsumgebung nutzen
 - › Klasse / Methode automatisch generieren
 - › Rote Tests auf einfachste Art grün bekommen
- › Test/Test möglichst häufig wiederholen

TDD: Ein erstes Beispiel

Schritt 1:

Die Methode „public static int calculate(String string)“ der Klasse „StringCalculator“ gibt für einen null-String den Wert 0 zurück.

So schreibe ich

```
public class StringCalculatorTest {  
  
    @Test  
    public void nullAsInputResultsIn0() {  
        int result = StringCalculator.calculate(string: null);  
        Assertions.assertEquals(expected: 0, result);  
    }  
}
```

„StringCalculator“ erstellen.

„calculate“ erstellen

Conzentration auf den Schritt

Test schreiben

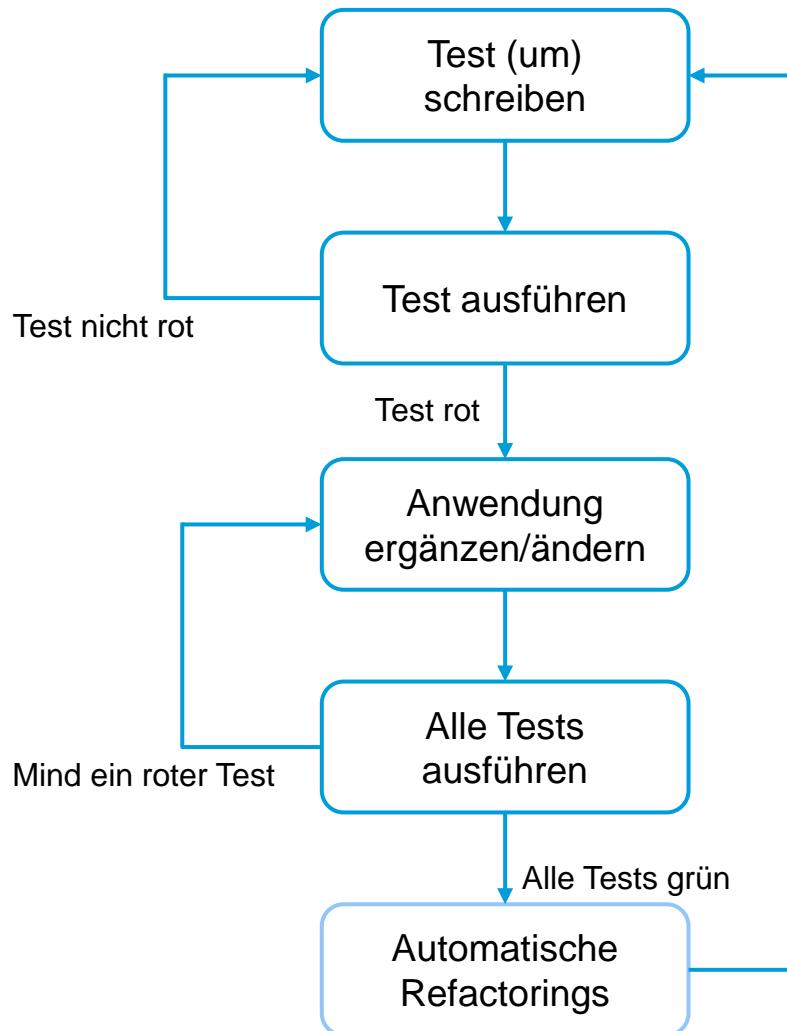
- › Der Test/die Tests testen genau (und nur das), was im jeweiligen Schritt beschrieben ist
- › Häufig kompiliert's dann im ersten Schritt nicht
 - › Perfekt, falscher geht's nicht ;)

Test grün bekommen

- › Features der Entwickler verstehen
 - › Klasse / Interface
- › Test/Testen möglichst häufig wiederholen

```
public class StringCalculator {  
    public static int calculate(String string) {  
        return 0;  
    }  
}
```

Grundprinzip von TDD





Ein leerer String gibt die Zahl 0 zurück



Ein leerer String gibt die Zahl 0 zurück

Ein String mit nur einer Zahl gibt diese Zahl zurück



- Ein leerer String gibt die Zahl 0 zurück
- Ein String mit nur einer Zahl gibt diese Zahl zurück
- Ein String mit zwei Komma-separierten Zahlen gibt die Summe dieser Zahlen zurück



- Ein leerer String gibt die Zahl 0 zurück
- Ein String mit nur einer Zahl gibt diese Zahl zurück
- Ein String mit zwei Komma-separierten Zahlen gibt die Summe dieser Zahlen zurück
- Ein String mit zwei Zahlen in jeweils einer separaten Zeile gibt die Summe der Zahlen zurück



- Ein leerer String gibt die Zahl 0 zurück
- Ein String mit nur einer Zahl gibt diese Zahl zurück
- Ein String mit zwei Komma-separierten Zahlen gibt die Summe dieser Zahlen zurück
- Ein String mit zwei Zahlen in jeweils einer separaten Zeile gibt die Summe der Zahlen zurück
- Ein String mit drei Zahlen jeweils separiert wie c) und d) gibt die Summe dieser Zahlen zurück



- Ein leerer String gibt die Zahl 0 zurück
- Ein String mit nur einer Zahl gibt diese Zahl zurück
- Ein String mit zwei Komma-separierten Zahlen gibt die Summe dieser Zahlen zurück
- Ein String mit zwei Zahlen in jeweils einer separaten Zeile gibt die Summe der Zahlen zurück
- Ein String mit drei Zahlen jeweils separiert wie c) und d) gibt die Summe dieser Zahlen zurück
- Ein String mit mindestens einer negativen Zahl wird eine Exception



- Ein leerer String gibt die Zahl 0 zurück
- Ein String mit nur einer Zahl gibt diese Zahl zurück
- Ein String mit zwei Komma-separierten Zahlen gibt die Summe dieser Zahlen zurück
- Ein String mit zwei Zahlen in jeweils einer separaten Zeile gibt die Summe der Zahlen zurück
- Ein String mit drei Zahlen jeweils separiert wie c) und d) gibt die Summe dieser Zahlen zurück
- Ein String mit mindestens einer negativen Zahl wirft eine Exception
- Eine Zahl größer 1000 wird ignoriert



Katas

- › Kleine wiederkehrende Übungen
 - Zur Perfektionssteigerung
 - Sicherung von Handgriffen
- › Teilnehmer
 - Alleine
 - In Gruppe
- › Falls Kata mehrschrittig ist:
 - Immer nur den jeweiligen Schritt lesen und durchführen
- › Im nur volle Konzentration auf den aktuellen Schritt bzw. die jeweilige Aufgabe
- › Ein Mitglied kommt nach vorne
 - Entweder auf Zeit
 - Oder für eine Teilaufgabe
- › Arbeit an dem Kata / dem Kata-Schritt
 - Beschreibt seine Arbeit
 - Wenn man nicht mehr weiter kommt, kann man sich helfen lassen
 - Nach Fertigstellung / Zum Ende wird das Ergebnis diskutiert

Quelle für Katas und Übungen:

<https://codeforces.org/>

- › Viele Übungen, meist mit dem Hintergrund der Erstellung optimierten Codes
 - Ziel ist die Erzeugung effizienter Algorithmen

Weitere Quelle für Katas:

<https://adventofcode.com/>

- › Ein wenig Zeitvertreib zu jeder Jahreszeit
- › <https://adventofcode.com/2015>
- › <https://adventofcode.com/2016>
- › <https://adventofcode.com/2017>
- › <https://adventofcode.com/2018>
- › <https://adventofcode.com/2019>

- › Es sind keine JAVA-Katas, aber Aufgaben, die mit Hilfe kleiner Programmen gelöst werden können



Übung 15

› Resourcen

de.hegmanns.training.junit5.practice15

Entwickelt Sie den Kennwortchecker für „gute“ Kennwörter:

Ein "gutes" Kennwort liegt dann vor, wenn der Passwortchecker keinen Fehler wirft.

Folgende Bedingungen müssen für ein "gutes" Kennwort vorliegen:

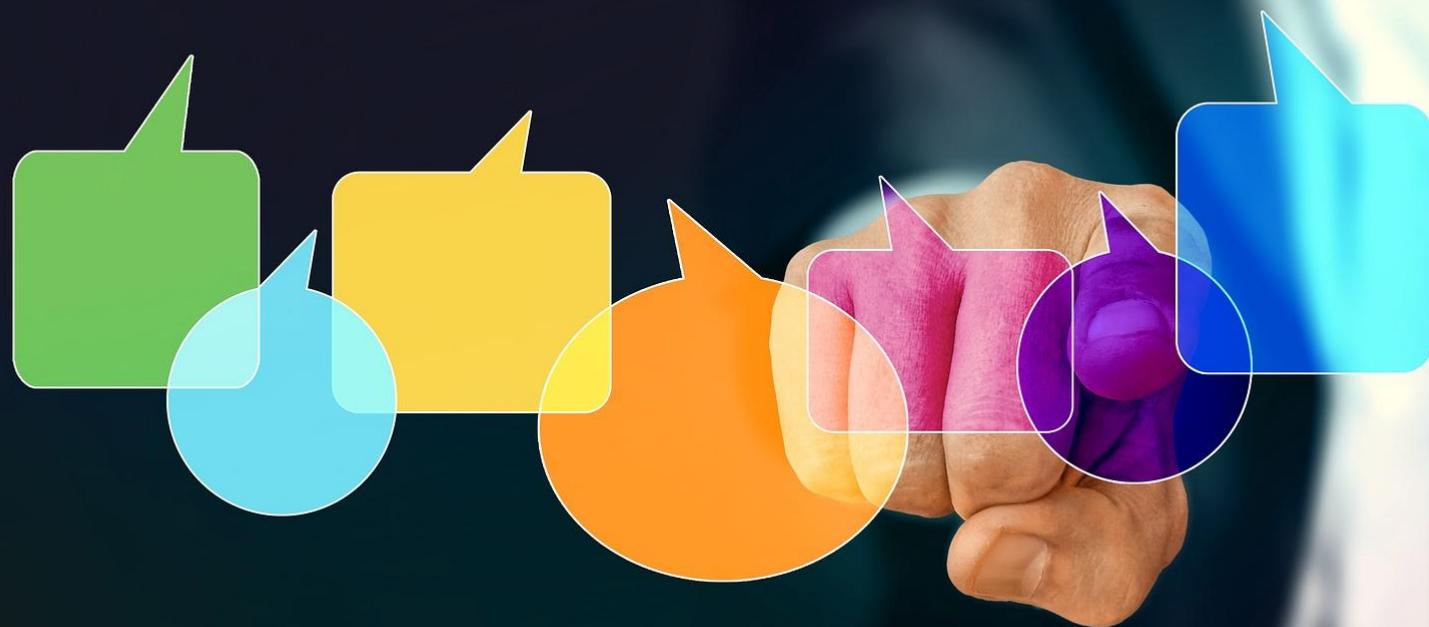
- Kennwort muss aus mehr als 8 Zeichen bestehen
- Kennwort muss aus mindestens einem groß geschriebenen Buchstaben bestehen
- Kennwort muss aus mindestens einem klein geschriebenen Buchstaben bestehen
- Kennwort muss aus mindestens einer Ziffer bestehen
- Kennwort muss aus mindestens einem der folgenden Zeichen bestehen: Leerzeichen, Komma, Punkt, Semikolon, Ausrufezeichen
- Falls nach einem groß geschriebenen Buchstaben noch ein Buchstabe kommt, muss dieser Buchstabe klein geschrieben sein
- Wenn nur die Buchstaben (egal ob klein oder groß) aus dem Kennwort extrahiert werden, darf Nicht das Wort "java" herauskommen.
(Klein- und Großschreibung soll hier ignoriert werden)

Ende des Tages ...



Kurzes Feedback des zweiten Tages

- › Übungslänge? (zu lang, zu kurz, richtige Länge)
- › Übungen? (zu einfach, zu kompliziert, genau richtig)
- › Geschwindigkeit? (zu schnell, zu langsam, richtig)
- › Themen? (fast alles bekannt, vieles bekannt, viel Neues)
- › Resümee?





Bis morgen