

- Motivation
- Testklassifikation
- Black Box Testtechniken
- White Box Testtechniken
- Testmetriken
- Grenzen des Software Tests
- ➡ ■ Testautomatisierung



Links und Literatur zu JUnit

- **Unfassende Darstellung (JUnit 4):**
M. Tamm: JUnit Profiwissen, 1. Auflage 2013, dpunkt.verlag
- **Bechold et al.: JUnit 5 User Guide:**
<https://junit.org/junit5/docs/current/user-guide/>
- **Gulati, Sharma: Java Unit Testing with JUnit 5**, apress, 2017
- **JUnit5 Tutorial:** <https://howtodoinjava.com/junit-5-tutorial/>
- **JUnit Website incl Tutorials:**
 - <http://junit.org/junit4/>
 - <https://junit.org/junit5/>
- **JUnit Artikel (JUnit 4)**
<http://www.vogella.com/tutorials/JUnit/article.html>
- **JUnit Tutorial (JUnit 4)**
<http://www.javacodegeeks.com/2014/11/JUnit-tutorial-unit-testing.html>
- **Einführung in JUnit3:** Kent Beck: JUnit Pocket Guide, Kindle Edition, O'Reilly

Achten Sie bei Tutorials und Büchern stets auf die gewünschte Version von JUnit!

Testautomatisierung



Motivation / Idee

- Ziele von JUnit
- Features von JUnit
- Einfaches Beispiel
- TestSuites
- TestFixturees
- JUnit Annotations
- JUnit Assertions
- Namenskonventionen
- Parametrisierbare Tests

- JUnit Tags
- JUnit 4
 - JUnit Rules
 - JUnit Theories
 - JUnit Custom Runners
- JUnit3 vs JUnit4 vs JUnit5



Beispiel: Testen der java.util.ArrayList

1. Versuch:

```
public static void firstArrayListTest(){  
    List <String> myArrayList= new ArrayList<String>();  
    String element = new String();  
    System.out.println(myArrayList.size());  
    myArrayList.add(element);  
    System.out.println(myArrayList.size());  
}
```

Prüfen und interpretieren!

Beispiel: Testen der java.util.ArrayList

2. Versuch:

```
public static void secondArrayListTest(){  
    List <String> myArrayList= new ArrayList<String>();  
    String element = new String();  
    System.out.println(myArrayList.size() == 0);  
    myArrayList.add(element);  
    System.out.println(myArrayList.size()==1);  
}
```

Prüfen!

Beispiel: Testen der java.util.ArrayList

3. Versuch:



Man kriegt mit, wenn der Test fehlschlägt.
➔ Keine manuelle Prüfung, sondern automatisierte Tests

```
public static void automatedArrayListTest(){  
    List <String> myArrayList= new ArrayList<String>();  
    String element = new String();  
    assertTrue(myArrayList.size()==0);  
    myArrayList.add(element);  
    assertTrue(myArrayList.size()==1);  
}
```

```
public static void assertTrue(boolean condition) {  
    if(!condition){  
        throw new RuntimeException("Assertion failed");  
    }  
}
```

Warum überhaupt automatisieren

→ Es gibt ganz viele Argumente.

Hauptargument:

Vertrauen in die eigene Arbeit



Zeitliche Vorteile der Testautomatisierung

- Kurzfristig für den Entwickler
 - Zeitersparnis bei Fehlerfinden und Korrigieren
- Langfristig für den Entwickler
 - Sicherheit, den Code langfristig warten zu können, ohne ihn zu brechen.
- Für das Team und den Kunden
 - Einfache Integration von gutgetestetem Code

Defect Entwicklung bei häufigen Tests

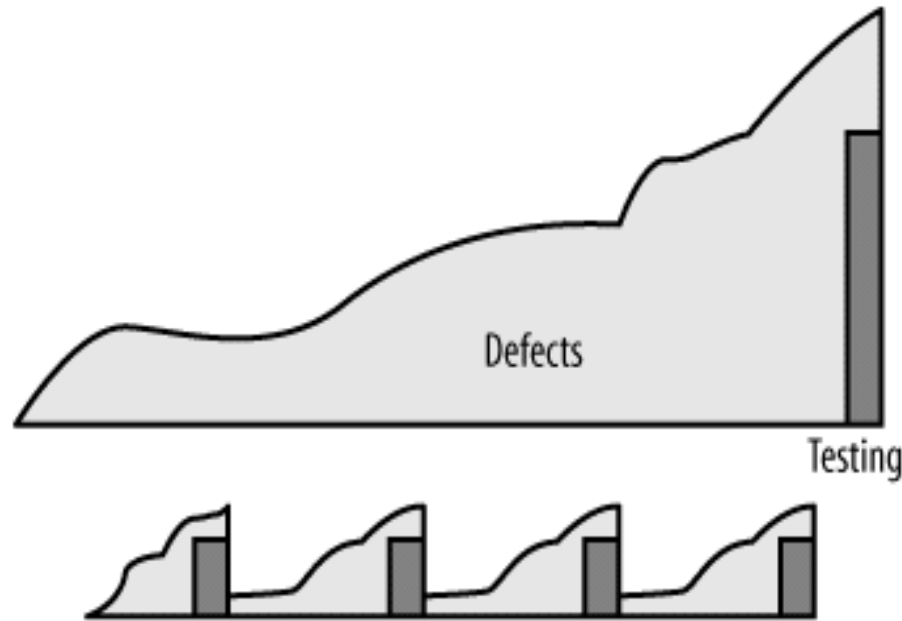


Figure 1-1. Frequent testing leaves fewer defects at the end

Quelle: *JUnit Pocket Guide*, Kent Beck, Kindle Edition



Testautomatisierung

- [Motivation / Idee](#)
- ➔ ▪ [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Tags](#)
- [JUnit 4](#)
 - [JUnit Rules](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4 vs JUnit5](#)



JUnit bietet eine Infrastruktur, um viele Tests automatisiert laufen zu lassen und das Ergebnis wiederzugeben.

JUnit

- Lässt Tests automatisiert laufen.
- Lässt viele Tests gemeinsam laufen und fasst das Ergebnis zusammen.
- Vergleicht Ergebnisse mit Erwartungen und teilt Unterschiede mit.

Ziele von JUnit

- Tests sollen einfach zu schreiben sein.
- Es soll einfach sein, das Schreiben von Tests zu lernen.
- Schnelle Testausführung.
- Einfache Testausführung (per Knopfdruck, einfache Darstellung der Ergebnisse).
- Isolierte Ausführung, keine Beeinflussung von Tests untereinander.
- Tests sollen zusammensetzbar sein.



Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- ➔ ▪ [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Tags](#)
- [JUnit 4](#)
 - [JUnit Rules](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4 vs JUnit5](#)



Features von JUnit

- Infrastruktur für automatisierte Tests
 - Test schreiben
 - Test durchführen
 - Test auswerten
- Test vorbereiten
- Test nachbereiten
- Tests organisieren
- Parametrisierbare Tests
- ...

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- ➔ ▪ [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Tags](#)
- [JUnit 4](#)
 - [JUnit Rules](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4 vs JUnit5](#)



Bsp. Calculator

Zu testende Klasse

```
package demopackage;  
  
public class Calculator {  
  
    public int add(int n, int m) {  
        return m + n;  
    }  
  
}
```



Bsp. Calculator

Testklasse

```
package demopackage;
```

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

```
import org.junit.jupiter.api.Test;
```

```
class MyFirstJUnitJupiterTests {
```

```
    private final Calculator calculator = new Calculator();
```

```
    @Test
```

```
    void addition() {
```

```
        assertEquals(2, calculator.add(1, 1)) ;
```

```
    }
```

```
}
```

Import der Assertions

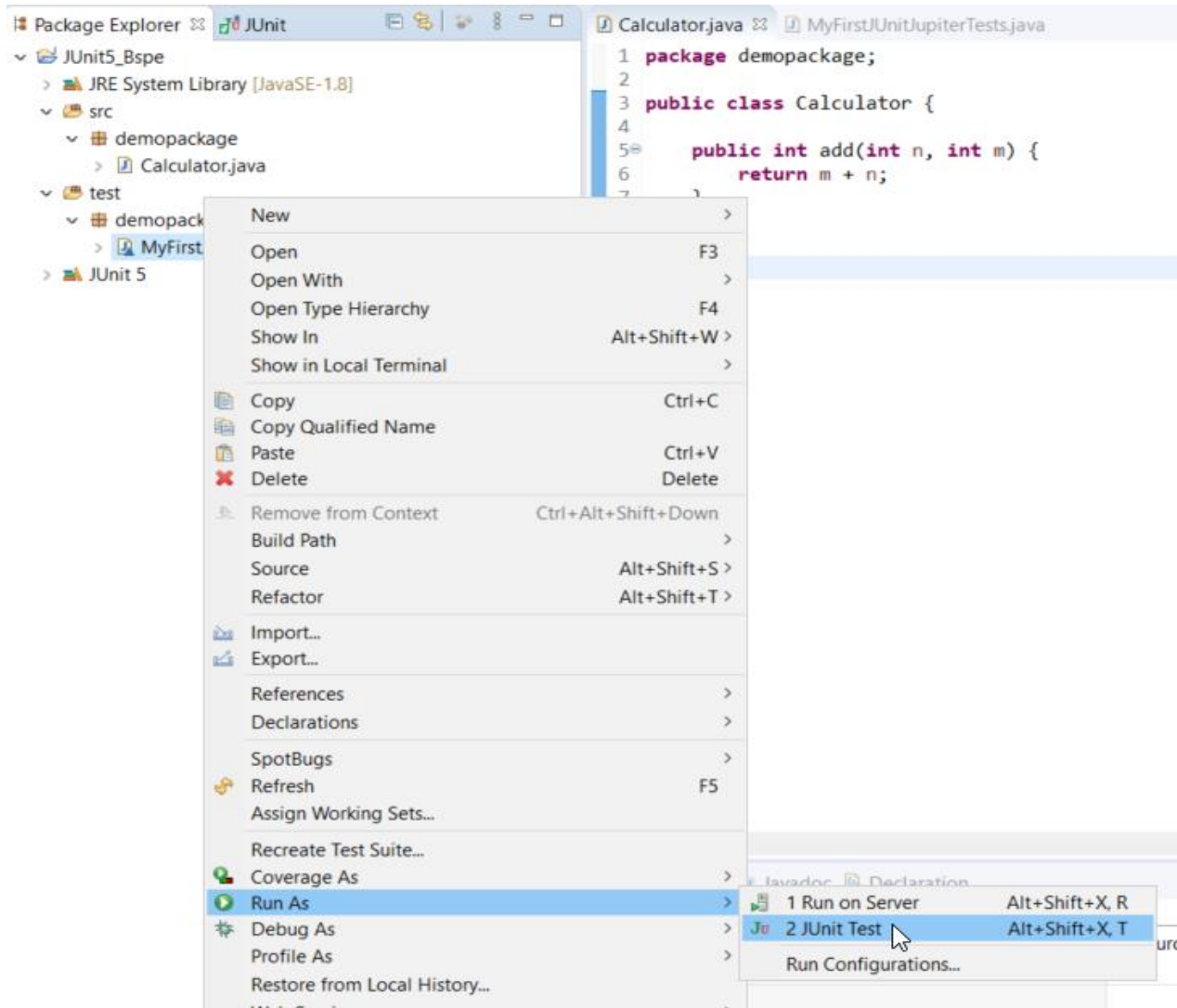
Import der Annotation

Beliebiger Name der Testklasse

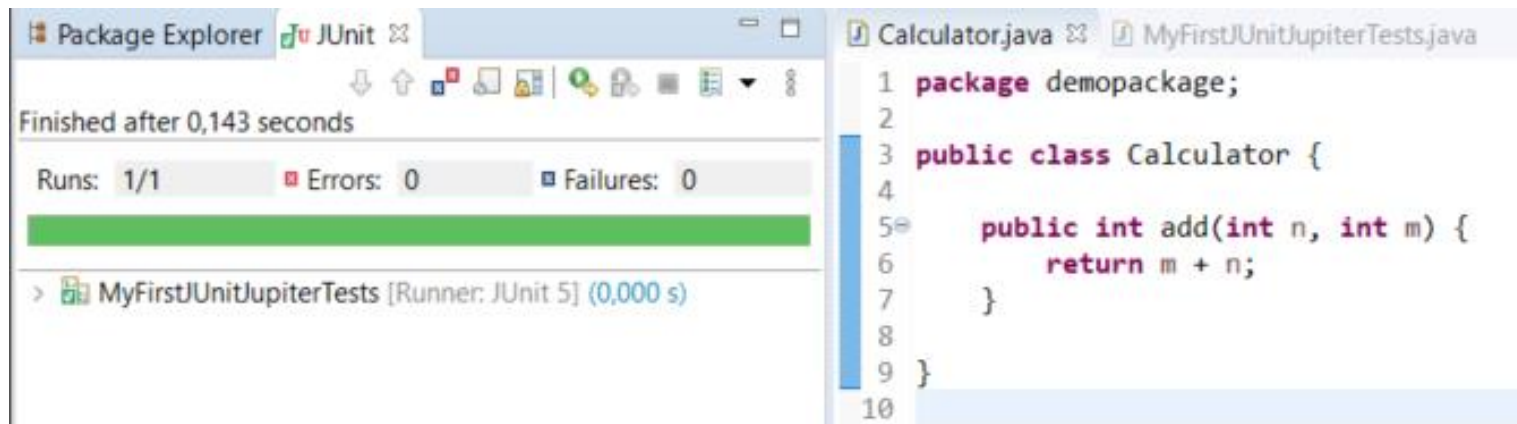
Kennzeichnung als Test

Assert Methode zur Überprüfung

Test laufen lassen (eclipse)



Ergebnis



The screenshot displays an IDE interface with two main panels. The left panel, titled 'JUnit', shows the results of a test run. It indicates that the test 'MyFirstJUnitJupiterTests' was completed after 0.143 seconds, with 1/1 runs, 0 errors, and 0 failures. A green progress bar is visible. The right panel shows the source code of the 'Calculator.java' file, which defines a 'Calculator' class with an 'add' method. The code is as follows:

```
1 package demopackage;
2
3 public class Calculator {
4
5     public int add(int n, int m) {
6         return m + n;
7     }
8
9 }
10
```



Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- ➔ ▪ [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)

- [JUnit Tags](#)
- [JUnit 4](#)
 - [JUnit Rules](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4 vs JUnit5](#)



Es kann sinnvoll sein, nur einen Teil der Tests laufen zu lassen (z.B. alle Smoketests, alle Tests zu einer Komponente etc).

→ Unterstützung durch IDE Konfigurationen
Oder

→ Erstellung von **JUnit TestSuites**

→ Standard Weg, unabhängig von einer IDE.

→ Test Suites können einfacher in einer Sourceverwaltung verwaltet werden.



Test Suites

JUnit 5: Möglichkeit Tests gemeinsam laufen zu lassen, die in verschiedenen Testklassen oder verschiedenen Packages liegen.

```
package demopackage;
```

```
import org.junit.platform.runner.JUnitPlatform;  
import org.junit.platform.suite.api.SelectClasses;  
import org.junit.runner.RunWith;
```

Runner für Suites

```
@RunWith(JUnitPlatform.class)  
@SelectClasses({MyFirstJUnitJupiterTests.class, SecondClassUnderTest.class})  
class TestSuiteExample {  
}
```

Annotation für Klassen
der Suite

Annotationen für Testsuites:

- `@SelectPackages()` : Führt die Tests in den angegebenen Packages und den Unterpackages aus.
 - `@IncludePackages()`: führt nur die angegebenen Packages aus.
 - `@ExcludePackages()`: führt alle außer den angegeben Packages aus
- `@SelectClasses()`: Führt die Tests der übergebenen Klassen aus.
- `@IncludeClassNamePatterns()`
- `@ExcludeClassNamePatterns`
- `@IncludeTags()`
- `@ExcludeTags()`

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
-  [TestFixtures](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Tags](#)
- [JUnit 4](#)
 - [JUnit Rules](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4 vs JUnit5](#)



Bsp: Test einer Client-Server-Kommunikation

```
public void testPing(){  
    Server server = new Server();  
    server.start();  
    Client client = new Client();  
    client.start();  
    client.send("ping");  
    assertEquals("ack", client.receive());  
    client.stop();  
    server.stop();  
}
```



Fixtures - Motivation

Bsp: Test einer Client-Server-Kommunikation, sauber

```
public void testPingsauber() {  
    Server server = new Server();  
    server.start();  
    try {  
        Client client = new Client();  
        client.start();  
        try {  
            client.send("ping");  
            assertEquals("ack", client.receive());  
        } finally {  
            client.stop();  
        }  
    } finally {  
        server.stop();  
    }  
}
```

Test vorbereiten

Eigentlicher Test

aufräumen

Fixtures

Deklaration

```
Server server;  
Client client;
```

Test setup

```
protected void bereiteTestVor(){  
    Server server = new Server();  
    server.start();  
    Client client = new Client();  
    client.start();  
}
```

Test Durchführung

```
public void testPing() {  
    client.send("ping");  
    assertEquals("ack", client.receive());  
}
```

Aufräumen

```
protected void raeumeTestauf(){  
    try{  
        client.stop();  
    }finally{  
        server.stop();  
    }  
}
```

Fixtures: Vorbereitungscode und Nachbereitungscode für Testmethoden.

➔ Initialisierung

Nutzen:

- Isolation des eigentliche Testcodes
- Vermeidung redundanten Vor- und Nachbereitungscode



Realisierung von Testfixtures durch Annotationen:

@BeforeEach

The annotated method will be run before each test method in the test class.

@AfterEach

The annotated method will be run after each test method in the test class.

@BeforeAll

The annotated method will be run before all test methods in the test class. This method must be static.

@AfterAll

The annotated method will be run after all test methods in the test class. This method must be static.



Wichtig: JUnit garantiert,

- alle `@After` Methoden werden immer aufgerufen werden, auch wenn eine davon eine Exception wirft.
- alle `@After` Methoden werden immer aufgerufen werden, auch wenn eine der `@Before` Methoden eine Exception wirft.



Bsp

Zu testende Klasse

```
package demopackage;

public class Calculator {

    private static int result;

    public void add(int n) {
        result = result + n;
    }

    public void subtract(int n) {
        result = result - 1;           //Bug : result = result - n
    }

    public void multiply(int n) {}     //Not implemented yet

    public void divide(int n) {
        result = result / n;
    }

    public void square(int n) {
        result = n * n;
    }

    public void clear() {               // Ergebnis löschen
        result = 0;
    }

    public void switchOn() {           // Bildschirm einschalten, Piepsen, oder was
        result = 0;                   // Taschenrechner halt so tun
    }
}
```

Bsp Testklasse

```
class LifecycleDemoTest {  
  
    private static Calculator calculator;  
  
    @BeforeAll  
    public static void switchOnCalculator() {  
        System.out.println("\tSwitch on calculator");  
        calculator = new Calculator();  
        calculator.switchOn();  
    }  
  
    @BeforeEach  
    public void clearCalculator() {  
        System.out.println("zu Beginn jeden Tests wird der Calculator zurueckgesetzt");  
        calculator.clear();  
    }  
}
```

```
    @AfterEach  
    void tearThis(){  
        System.out.println("@AfterEach executed");  
    }  
  
    @AfterAll  
    public static void switchOffCalculator() {  
        System.out.println("\tSwitch off calculator");  
        calculator.switchOff();  
        calculator = null;  
    }  
}
```

```
@Disabled("not ready yet")  
@Test  
public void test_multiply() {  
    System.out.println("test_multiply()");  
    calculator.add(10);  
    calculator.multiply(10);  
    assertEquals(calculator.getResult(), 100);  
}
```


Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixtures](#)
-  [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Tags](#)
- [JUnit 4](#)
 - [JUnit Rules](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4 vs JUnit5](#)



Test Annotations

ANNOTATION	DESCRIPTION
<code>@BeforeEach</code>	The annotated method will be run before each test method in the test class.
<code>@AfterEach</code>	The annotated method will be run after each test method in the test class.
<code>@BeforeAll</code>	The annotated method will be run before all test methods in the test class. This method must be static.
<code>@AfterAll</code>	The annotated method will be run after all test methods in the test class. This method must be static.
<code>@Test</code>	It is used to mark a method as junit test
<code>@DisplayName</code>	Used to provide any custom display name for a test class or test method
<code>@Disable</code>	It is used to disable or ignore a test class or method from test suite.
<code>@Nested</code>	Used to create nested test classes
<code>@Tag</code>	Mark test methods or test classes with tags for test discovering and filtering
<code>@TestFactory</code>	Mark a method is a test factory for dynamic tests

Aus <https://howtodoinjava.com/junit-5-tutorial/#annotations>

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- ➔ ▪ [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Tags](#)
- [JUnit 4](#)
 - [JUnit Rules](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4 vs JUnit5](#)



Assertions

- assertEquals()
- assertNotEquals()
- assertEqualsArrayEquals()
- assertEqualsIterableEquals()
- assertEqualsLinesMatch()
- assertNotNull()
- assertNull()
- assertNotSame()
- assertSame()
- assertEqualsTimeout()
- assertEqualsTimeoutPreemptively()
- assertTrue()
- assertFalse()
- assertEqualsThrows()
- fail()

Beispiele unter <https://howtodoinjava.com/junit5/junit-5-assertions-examples/>

Testautomatisierung

- [Motivation / Idee](#)
 - [Ziele von JUnit](#)
 - [Features von JUnit](#)
 - [Einfaches Beispiel](#)
 - [TestSuites](#)
 - [TestFixturees](#)
 - [JUnit Annotations](#)
 - [JUnit Assertions](#)
 - [Namenskonventionen](#)
 - [Parametrisierbare Tests](#)
- [JUnit Tags](#)
 - [JUnit 4](#)
 - [JUnit Rules](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
 - [JUnit3 vs JUnit4 vs JUnit5](#)



JUnit Konventionen

- Trennung Code unter Test von Testcode

JUnit5_Bspe

JRE System Library [JavaSE-1.8]

src

demopackage

Calculator.java

Fibonacci.java

SecondClassUnderTest.java

test

demopackage

FibonacciTest.java

LifeCycleDemoTest.java

MyFirstJUnitJupiterTests.java

SecondTest.java

TagDemoTest.java

TestSuiteExample.java

UsingTagsTest.java

Code under Test

Testcode



JUnit Namenskonventionen

Klasse, die eine andere Klasse testet, hat den Namen der zu testenden Klasse + „Test“

Bsp:

- zu testen: *Car.java*
- Testklasse: *CarTest.java*



JUnit Namenskonventionen

Test Methoden:

Konvention (nicht zwingend seit JUnit 4): Beginne den Namen mit „*test*“

Benennung (Konvention nach *M. Tamm: JUnit Profiwissen*):

- *test()*
- *test_<Name der getesteten Methode>()*
- *test_that_<erwartetes Verhalten>()*
- *test_that_<erwartetes Verhalten>_when_<Vorbedingung>()*



Testautomatisierung

- [Motivation / Idee](#)
 - [Ziele von JUnit](#)
 - [Features von JUnit](#)
 - [Einfaches Beispiel](#)
 - [TestSuites](#)
 - [TestFixturees](#)
 - [JUnit Annotations](#)
 - [JUnit Assertions](#)
 - [Namenskonventionen](#)
 - [Parametrisierbare Tests](#)
- [JUnit Tags](#)
 - [JUnit 4](#)
 - [JUnit Rules](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
 - [JUnit3 vs JUnit4 vs JUnit5](#)



Parametrisierter Test → Instanzen für das Kreuzprodukt aus Test Daten Elementen und Testmethoden.

Bsp: Klasse zu testen: Fibonacci.java

```
package demo;  
  
public class Fibonacci {  
    public static int compute(int n) {  
        int result = 0;  
  
        if (n <= 1) {  
            result = n;  
        } else {  
            result = compute(n - 1) + compute(n - 2);  
        }  
  
        return result;  
    }  
}
```



Testen verschiedener Kombinationen

Bisher:

```
package demopackage;

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

class FibonacciTest {

    @Test
    void testCompute() {
        assertEquals(0, Fibonacci.compute(0));
        assertEquals(1, Fibonacci.compute(1));
        assertEquals(1, Fibonacci.compute(2));
        assertEquals(2, Fibonacci.compute(3));
        assertEquals(3, Fibonacci.compute(4));
        assertEquals(5, Fibonacci.compute(5));
        assertEquals(8, Fibonacci.compute(6));
    }
}
```

Testen verschiedener Kombinationen - besser

```
@ParameterizedTest
@CsvSource({
    "0,0",
    "1,1",
    "1,2",
    "2,3",
    "3,4",
    "5,5",
    "8,6"
})
void testWithCsvSource(int result, int input) {
    System.out.println("Test mit " + result + " , " + input);
    assertEquals(result, Fibonacci
        .compute(input));
}
```

Parametrisierter Test

Daten, hier als csv Values

Verwendung der Daten im
Test

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)



- [JUnit Tags](#)
- [JUnit 4](#)
 - [JUnit Rules](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4 vs JUnit5](#)



Tags

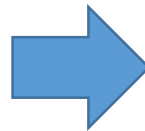
Mit der JUnit5 @Tag Annotation können Sie Filter für Testpläne setzen.

Tags definieren

```
@Test
@Tag("Tagdemo")
void testCompute_1() {
    assertEquals(1, Fibonacci.compute(1));
}

@Test
@Tag("Tagdemo")
void testCompute_2() {
    assertEquals(1, Fibonacci.compute(2));
}

@Test
@Tag("Tagdemo")
void testCompute_3() {
    assertEquals(3, Fibonacci.compute(4));
}
```



Tags nutzen

```
package demopackage;

import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.IncludeTags;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectPackages("demopackage")
@IncludeTags("Tagdemo")
class UsingTagsTest {
}
```

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)



- [JUnit Tags](#)
- [JUnit 4](#)
 - [JUnit Rules](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4 vs JUnit5](#)



Motivation:


- @Before und @After wird verwendet, um Tests vorzubereiten und nachher aufzuräumen.
- Bsp: temporäres Verzeichnis anlegen, nachher löschen.
 - ➔ das braucht man öfter als nur in einer Testklasse

➔ Was tun?



JUnit4 Rules

Lösungsmöglichkeiten:

1. Redundante @Before und @After Methoden in verschiedenen Klassen 
2. Basisklasse für alle Testklassen mit gemeinsam zu nutzenden @Before und @After Methoden 
3. Hierarchie von Testklassen mit Kombinationen von @Before und @After Methoden 

oder

JUnit4 Rules



JUnit4 Rules

- JUnit4 Rules erlauben es, querschnittliche Vorbereitungs- und Aufräumarbeiten nur einmal zu implementieren und zu pflegen.
- Es gibt bereits vorgefertigte JUnit4 Rules, die mit JUnit4 ausgeliefert werden.



JUnit4 Rules Beispiel

```
public class AssetManagerTest {  
  
    @Rule  
    public TemporaryFolder tempFolder = new TemporaryFolder();  
  
    @Test  
    public void countsAssets() throws IOException {  
        AssetManager am = new AssetManager();  
        File assets = tempFolder.newFolder("assets");  
        am.createAssets(assets, 3);  
        assertEquals(3, am.countAssets());  
    }  
}
```

Zu TemporaryFolder siehe

<http://junit.org/junit4/javadoc/latest/org/junit/rules/TemporaryFolder.html>



Um eine Testklasse mit einem als MethodRule Klasse programmierten Testaspekt auszustatten:

- Öffentliche Membervariable zur Testklasse.
- Diese mit @Rule annotieren
- Der Variablen eine neue Instanz der gewünschten MethodRule Klasse zuweisen.



Diese Rules gibt es bereits:

- TemporaryFolder Rule
- ExternalResource Rules
- ErrorCollector Rule
- Verifier Rule
- TestWatchman/TestWatcher Rules
- TestName Rule
- Timeout Rule
- ExpectedException Rules

*Teilweise als
Superklassen für selbst
zu implementierende
Klassen*



Siehe <https://github.com/junit-team/junit4/wiki/Rules>

Entsprechungen (JUnit4)

@Before	@Rule
@BeforeClass	@ClassRule



JUnit4 RuleChain

RuleChain erlaubt es, Rules hintereinander auszuführen.

```
public static class UseRuleChain {  
    @Rule  
    public TestRule chain = RuleChain  
        .outerRule(new LoggingRule("outer rule"))  
        .around(new LoggingRule("middle rule"))  
        .around(new LoggingRule("inner rule"));  
  
    @Test  
    public void example() {  
        assertTrue(true);  
    }  
}
```



```
starting outer rule  
starting middle rule  
starting inner rule  
finished inner rule  
finished middle rule  
finished outer rule
```



Eigene Rules schreiben:

- Klasse anlegen, die das Interface TestRule implementiert:

<http://junit.org/junit4/javadoc/latest/org/junit/rules/TestRule.html>

```
public interface TestRule {  
    /**  
     * Modifies the method-running {@link Statement} to implement this  
     * test-running rule.  
     *  
     * @param base The {@link Statement} to be modified  
     * @param description A {@link Description} of the test implemented in {@code base}  
     * @return a new statement, which may be the same as {@code base},  
     *         a wrapper around {@code base}, or a completely new Statement.  
     */  
    Statement apply(Statement base, Description description);  
}
```



Custom Rules

```
public interface TestRule {  
    /**  
     * Modifies the method-running {@link Statement} to implement this  
     * test-running rule.  
     *  
     * @param base The {@link Statement} to be modified  
     * @param description A {@link Description} of the test implemented in {@code base}  
     * @return a new statement, which may be the same as {@code base},  
     *         a wrapper around {@code base}, or a completely new Statement.  
     */  
    Statement apply(Statement base, Description description);  
}
```

Repräsentiert den Aufruf
der eigentlichen @Test
Methode (incl aller before
und after Methoden)

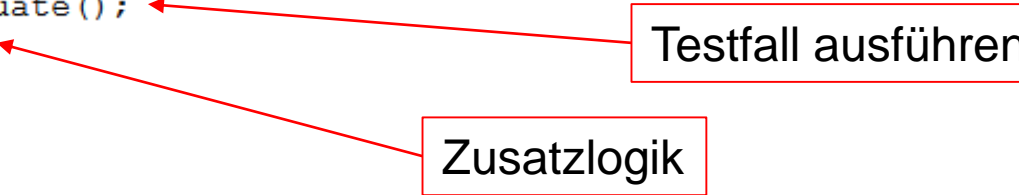
Weitere Beschreibung des Tests

```
public abstract class Statement {  
    /**  
     * Run the action, throwing a {@code Throwable} if anything goes wrong.  
     */  
    public abstract void evaluate() throws Throwable;  
}
```

Hier wird der eigentliche Test ausgeführt

Custom Rules – Bsp Verifier

```
public abstract class Verifier implements TestRule {  
    public Statement apply(final Statement base, Description description) {  
        return new Statement() {  
            @Override  
            public void evaluate() throws Throwable {  
                base.evaluate();  
                verify();  
            }  
        };  
    }  
  
    /**  
     * Override this to add verification logic. Overrides should throw an  
     * exception to indicate that verification failed.  
     */  
    protected void verify() throws Throwable {  
    }  
}
```



Oft verwendetes Muster: Erzeugen und Rückgabe eines anonymen inneren Statements, das in der evaluate Methode denjenigen Code enthält, der den Test ausmacht und an passender Stelle base.evaluate aufruft.

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)



- [JUnit Tags](#)
- [JUnit 4](#)
 - [JUnit Rules](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4 vs JUnit5](#)



Idee:

Allgemeingültige Aussage als Test formulieren
und durch eine Reihe von Testwerten
überprüfen.

Paper dazu:

<http://web.archive.org/web/20110608210825/http://shareandenjoy.saff.net/tdd-specifications.pdf>

Titel:

The Practice of Theories: Adding “For-all” Statements to “There-Exists” Tests



Beispiel: Test einer Multiplikation von BigInteger

Variante 1:

```
@Test
public void testMultiplyBigInteger() {
    assertEquals(new BigInteger("20"), new BigInteger("10").multiply(new BigInteger("2")));
    assertEquals(new BigInteger("30"), new BigInteger("10").multiply(new BigInteger("3")));
    assertEquals(new BigInteger("50"), new BigInteger("25").multiply(new BigInteger("2")));
    assertEquals(new BigInteger("2"), new BigInteger("1").multiply(new BigInteger("2")));
}
```

Test von Berechnung mit bestimmten Eingabewerten gegen Erwartungswerte der Berechnung für diese Eingabewerte.



Beispiel: Test einer Multiplikation von BigInteger

Variante 2:

```
@Test
public void testGeneral(){
    testMultiplyInGeneral(10,2);
    testMultiplyInGeneral(10,3);
    testMultiplyInGeneral(25,2);
    testMultiplyInGeneral(1,2);
}

public void testMultiplyInGeneral(int firstInt, int secondInt){
    System.out.println(firstInt + " und " + secondInt);
    assertEquals(new BigInteger(String.valueOf(firstInt*secondInt)),
        new BigInteger(String.valueOf(firstInt)).multiply(new BigInteger(String.valueOf(secondInt))));
}
```

Formulierung der Testmethode als allgemeine „Theorie“, Test mit im Prinzip unendlich vielen Werten möglich. Keine explizite Vorgabe des erwarteten Ergebnisses als Wert!

→ Diese Idee wird durch die Theories unterstützt.



Beispiel: Test einer Multiplikation von BigInteger

Variante 3:

```
@Theory
public void multiply_behaves_like_primitive(int firstInt, int secondInt){
    System.out.println("Test multiply_behaves_like_primitive() mit");
    System.out.println(firstInt + " und " + secondInt);
    assertEquals(new BigInteger(String.valueOf(firstInt*secondInt)),
        new BigInteger(String.valueOf(firstInt)).multiply(new BigInteger(String.valueOf(secondInt))));
}
```

```
@DataPoints
public static int[] INT_DATA = {1,2,3,4,5,6,7,8,9};
```

Umsetzung des Prinzips mithilfe von Theories.

Bsp zu JUnit Theories

```
@RunWith(Theories.class)
public class BigIntegerTest {

    @Theory
    public void multiply_follows_commutative(BigInteger i1, BigInteger i2) {
        assertEquals(i1.multiply(i2), i2.multiply(i1));
    }

    @Theory
    public void divide_is_inverse_of_multiply(BigInteger i1, BigInteger i2) {
        assumeThat(i2.toString(), is(not("0")));
        assertEquals(i1.multiply(i2).divide(i2), i1);
    }

    @DataPoints
    public static BigInteger[] TEST_DATA = new BigInteger[]{
        new BigInteger("-1"),
        new BigInteger("0"),
        new BigInteger("42"),
        new BigInteger("12121234343434231213"),
    };
}
```


Was ist an dem Beispiel besonders?

- Die Testmethoden haben mindestens einen Eingabeparameter.
- Es werden allgemeingültige Aussagen als Testmethoden formuliert.
- Der eigentliche Test läuft dann mit einem Set an Parametern durch, die man als @DataPoints annotiert.



- Die Testmethode und die Testdaten sind voneinander getrennt.
- Eine mit `@DataPoints` oder mit `@DataPoint` annotierte Variable oder statische Methode liefert ein Array oder einen Wert als Eingabewerte für einen bestimmten Datentyp.
- Der Theories Runner sammelt alle Beispielwerte ein unmittelbar bevor er die Theories Methode ausführt. Also nach allen `@BeforeClass` und `@Before` Methoden.



Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)



- [JUnit Tags](#)
- [JUnit 4](#)
 - [JUnit Rules](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4 vs JUnit5](#)



JUnit4 Runner:

- *BlockJUnit4ClassRunner*: Default Runner
- *Suite*: Standard Runner um TestSuites laufen zu lassen
- *Parameterized*: Runner für parametrisierte Tests
- *Categories*: Standard Runner für subsets von Tests, die entsprechend getagged sind.
- *Theories*: Runner für Testtheorien



Eigene Test Runner

Test Runner sind für die Ausführung der Tests zuständig.

Um sie zu verwenden, ist die Annotation `@RunWith()` nötig.

Testrunners Verantwortung:

- Testklassen Instanziierung
- Test Ausführung
- Reporting der Test Resultate




Test Runner

- Ein Test Case kann den Runner selbst angeben:
`@RunWith` Annotation
- Normalerweise genügen die bereits vorhandenen Runner. Wenn man selbst einen schreiben will, siehe
 - Michael Tamm: JUnit Profiwissen, dpunkt.verlag
 - <http://www.mscharhag.com/java/understanding-junits-runner-architecture>



Testautomatisierung

- [Motivation / Idee](#)
 - [Ziele von JUnit](#)
 - [Features von JUnit](#)
 - [Einfaches Beispiel](#)
 - [TestSuites](#)
 - [TestFixtures](#)
 - [JUnit Annotations](#)
 - [JUnit Assertions](#)
 - [Namenskonventionen](#)
 - [Parametrisierbare Tests](#)
- 
- [JUnit Tags](#)
 - [JUnit 4](#)
 - [JUnit Rules](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
 - [JUnit3 vs JUnit4 vs JUnit5](#)



JUnit 3

- Sehr weit verbreitet: JUnit4
- Aktuellste Version: JUnit5
- In vielen Projekten noch verwendet: JUnit3

```
package demopackage;

import junit.framework.TestCase;

public class CalculatorTest extends TestCase {

    Calculator calculator;

    protected void setUp() throws Exception {
        System.out.println("\tSwitch on calculator");
        calculator = new Calculator();
        calculator.switchOn();
        System.out.println("zu beginn jeden Tests wird der Calculator zuruecgesetzt");
        calculator.clear();
    }

    protected void tearDown() throws Exception {
        System.out.println("\tSwitch off calculator");
        calculator.switchOff();
        calculator = null;
    }

    public void testAdd() {
        calculator.add(1);
        calculator.add(1);
        assertEquals(calculator.getResult(), 2);
    }
}
```



JUnit3 Testcase – Unterschiede zu JUnit4

```
package demopackage;
```

```
import junit.framework.TestCase;
```

```
public class CalculatorTest extends TestCase {
```

```
    Calculator calculator;
```

```
    protected void setUp() throws Exception {  
        System.out.println("\tSwitch on calculator");  
        calculator = new Calculator();  
        calculator.switchOn();  
        System.out.  
            .println("zu Beginn jeden Tests wird der Calculator zuruecgesetzt");  
        calculator.clear();  
    }
```

```
    protected void tearDown() throws Exception {  
        System.out.println("\tSwitch off calculator");  
        calculator.switchOff();  
        calculator = null;  
    }
```

```
    public void testAdd() {  
        calculator.add(1);  
        calculator.add(1);  
        assertEquals(calculator.getResult(), 2);  
    }
```

```
}
```

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

Ableitung von `TestCase`

Methode `setUp()` und `tearDown()`
Statt Annotierte Methoden

Name muss mit „test“ beginnen

TestSuites in JUnit3

```
package meinpackage;

import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests extends TestSuite
{
    public static Test suite()
    {
        TestSuite mySuite = new TestSuite( "Meine Test-Suite" );
        mySuite.addTestSuite( meinpackage.MeineKlasseTest.class );
        // ... weitere Testklassen hinzufügen
        return mySuite;
    }
}
```



JUnit4 vs Junit5 - Annotations

FEATURE	JUNIT 4	JUNIT 5
Declare a test method	<code>@Test</code>	<code>@Test</code>
Execute before all test methods in the current class	<code>@BeforeClass</code>	<code>@BeforeAll</code>
Execute after all test methods in the current class	<code>@AfterClass</code>	<code>@AfterAll</code>
Execute before each test method	<code>@Before</code>	<code>@BeforeEach</code>
Execute after each test method	<code>@After</code>	<code>@AfterEach</code>
Disable a test method / class	<code>@Ignore</code>	<code>@Disabled</code>
Test factory for dynamic tests	NA	<code>@TestFactory</code>
Nested tests	NA	<code>@Nested</code>
Tagging and filtering	<code>@Category</code>	<code>@Tag</code>
Register custom extensions	NA	<code>@ExtendWith</code>

Aus <https://howtodoinjava.com/junit5/junit-5-vs-junit-4/>



JUnit4 vs JUnit5

Architektur

JUnit4: einzelnes jar file

JUnit5: Junit Platform, Junit Jupiter, Junit Vintage

JDK

JUnit4: Java 5 oder höher

JUnit5: Java 8 oder höher

Tagging/Filtering

JUnit4: @Category

JUnit5: @Tag

Suites

JUnit4: @RunWith und @Suite

JUnit5: @RunWith, @SelectPackages, @SelectClasses

Erweiterungen

JUnit4: Rules und Runners

JUnit5: Extensions

