



JUnit 5 Refresher Short

**More fun and less stomach ache during
development through clever tests**

Michael Inden



Agenda



- **PART 1: Why to test and good habits**
 - **PART 2: JUnit 5 Intro**
 - Architecture
 - Testing exceptions
 - **PART 3: JUnit 5 Advanced**
 - Parameterized Tests
 - Parameterized Tests Advanced
 - **PART 4: Test Smells**
 - **PART 5: Characterization Testing**
-



PART 1:

Why to test and good habits



What is testing?



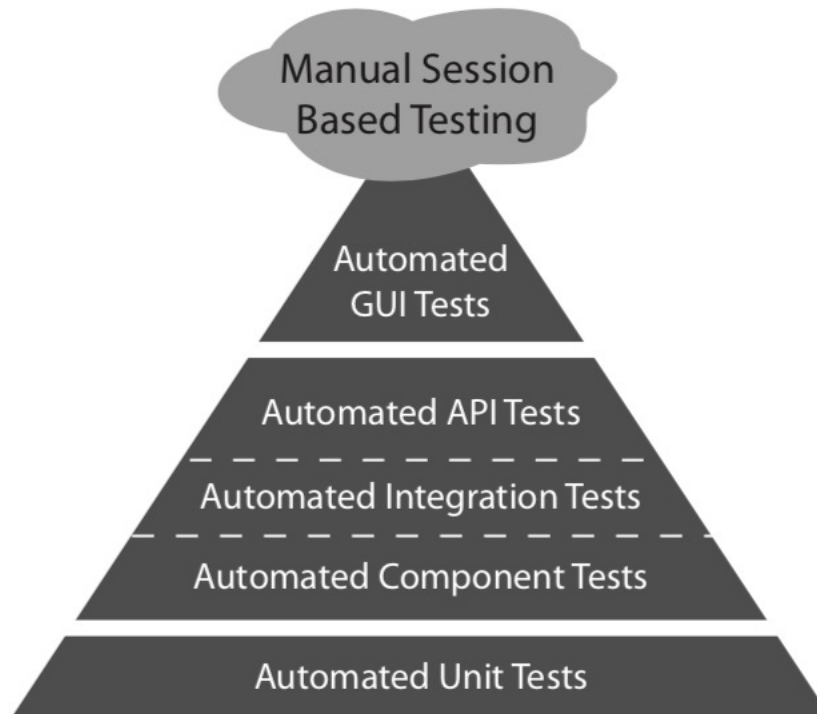
- Testing is the process of comparing the **actual behavior** of a program or a part of it (actual) with the **required behavior** (target).
 - Accordingly, testing does not correspond to the one-time start of a program with a few arbitrary operating actions.
 - to test behavior of the software under different conditions.
 - to act a little "maliciously" in order to uncover hidden problems or to provoke misbehavior.
-

Why do we test?

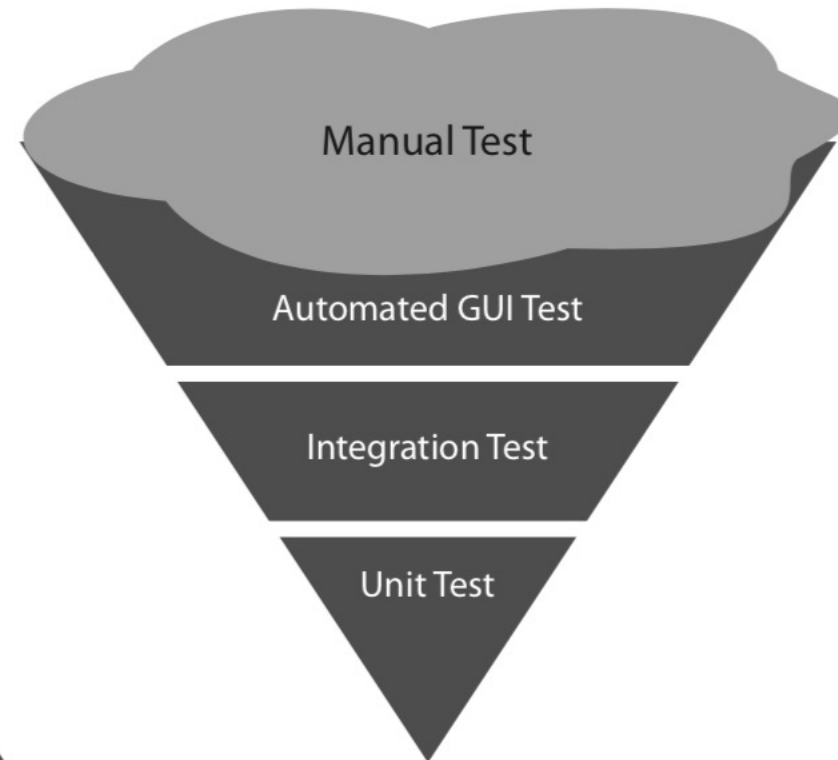


- Describe desired behavior
 - Check functionality (also **edge cases**)
 - Build a **safety net**
 - Quality assurance
 - Customer satisfaction
 - **less hassle, less nerves and more fun**
-

Test pyramid



The Ideal Testing
Automation Pyramid



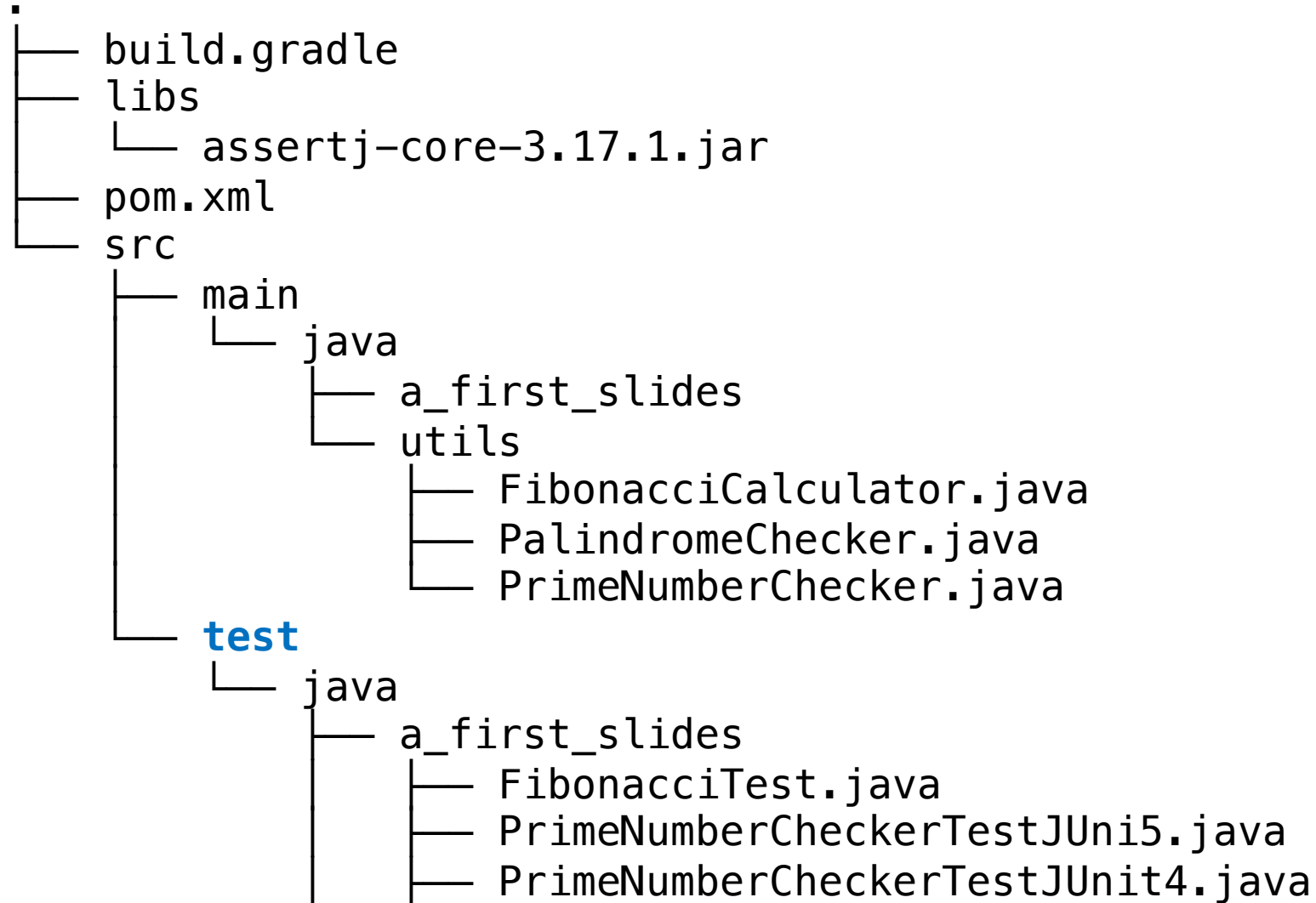
The Non-Ideal Testing
Automation Inverted Pyramid



Good habits



Maven Project Structure



Define test cases



- Unit tests **verify small components**, mostly classes or methods
 - **As isolated as possible** and without interaction with other components
 - Tests are **implemented in the form of methods**
 - Ideally: at **least one test method for each relevant application method**
 - A test method tests exactly one functionality (or only a part of it) , **ideally only 1 ASSERT !**
 - Keep test methods **short, clear and understandable**
 - BUT: How to achieve this?
-

Naming



- class **Abc** => associated test class **AbcTest**
- methods:
 - Optional: abbreviation test as start
 - Meaningful description of the test case:
 - Code method name, conditions and result in the name => **CamelCase often becomes unreadable**
 - Testing Guru Roy Osherove suggests the following:

MethodName_StateUnderTest_ExpectedBehavior

MethodName_ExpectedBehavior_WhenTheseConditions

calcSum_WithValidInputs_ShouldSumUpAllValues()

calcSum_ThrowsException_WhenNullInput()

Example: A first unit test with JUnit 5



- Test cases get written in the form of special test methods marked with the annotation `@Test`

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

public class A_FirstTestWithJUnit5
{
    @Test
    void assertMethodsInAction()
    {
        String expected = "Tim";
        String actual = "Tim";

        assertEquals(expected, actual);
        assertEquals(expected, "XYZ", "Message if comparison fails");
    }
}
```

AAA-Style



- **ARRANGE - ACT – ASSERT** (Also called GWT for GIVEN - WHEN - THEN)
- **ARRANGE:** preconditions and initializations (**test fixture**)
- **ACT:** then an action is executed
- **ASSERT:** check if the expected state has occurred

```
@Test
void listAdd_AAASStyle()
{
    // GIVEN: An empty list
    final List<String> names = new ArrayList<>();

    // WHEN: adding 2 elements
    names.add("Tim");
    names.add("Mike");

    // THEN: list should contain 2 elements
    assertEquals(2, names.size(), "list should contain 2 elements");
}
```

FAIR - Desired properties of unit tests



F – Fast, Focussed

A - Automated

I - Isolated

R – Reliable, Repeatable



What makes a good unit test?



A good Unit Test should be:



Easy
to write



Simple
to read



Trivial
to maintain

Unit Test Features: API Design & Documentation



- When writing unit tests, **design decisions** such as those regarding **cohesion, coupling and the design** of the API also play a role.
 - By implementing test cases, you use the API of your own classes, which makes it easier to **judge whether the interfaces provided are useful and manageable**.
 - Unit tests can thus **uncover possible weaknesses** in the APIs addressed by the tests before they are used in other components **and ensure more successful APIs**.
 - **Unit tests as documentation of the expected program behavior**
 - Documentation is **automatically always up to date**, otherwise the test cases would fail.
-



PART 2:

JUnit 5 Intro



JUnit 5

5 JUnit 5

[JUnit 4](#)

The new major version of the programmer-friendly
testing framework for Java

 User Guide

 Javadoc

 Code & Issues

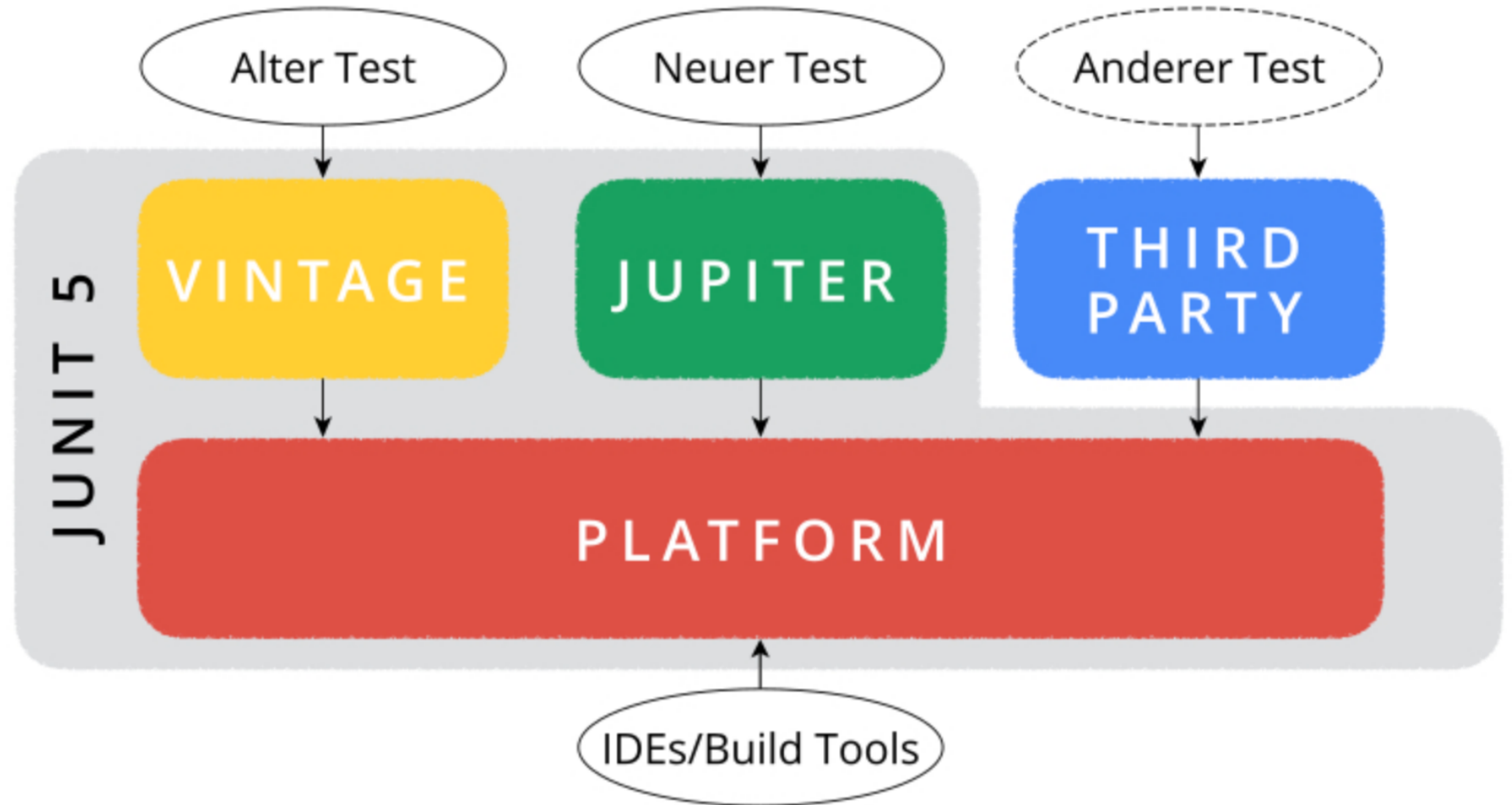
 Q & A

 Support JUnit

Architecture



- JUnit 5 =
JUnit Platform +
JUnit Jupiter +
JUnit Vintage



Assertions - check conditions



- Evaluation of conditions - The Assert (JUnit4) / Assertions (JUnit 5) class provides a set of verification methods that can be used to express conditions and thereby verify assertions about the source code under test:
 - **assertEquals()** – check two objects for equality of content (call **equals(Object)**) or two variables of primitive type for equality*
 - **assertTrue()** and **assertFalse()** – check boolean conditions
 - **assertNull()** and **assertNotNull()** – check object references for == null and != null respectively
 - **assertSame()** and **assertNotSame()** check object references for == or !=
 - **fail()** – deliberately make a test case fail

*) pay attention for floating point: float and double

A second unit test with JUnit 5



- Test cases are written in the form of special test methods marked with the annotation `@Test`

```
@Test
void assertMethodsInAction()
{
    String expected = "Tim";
    String actual = "Tim";
    assertEquals(expected, actual);
    assertEquals(expected, "XYZ", "Hint if wrong");

    assertTrue(true);
    assertTrue(true, "Always true");

    assertFalse(false);
    assertNull(null);
    assertNotNull(new Object());
    assertSame(null, null);
    assertNotSame(null, new Object());
}
```

Special test names



- With JUnit 4 one was limited to valid method names
- Special test names are now possible using the annotation **@DisplayName**

```
@Test
@DisplayName("(1 * 2 * 3) / 4 = 6/4 = 3/2 = 1.5")
void divideResultOfMultiplication()
{
    BigDecimal newValue = BigDecimal.ONE.multiply(BigDecimal.valueOf(2)).
                                         multiply(BigDecimal.valueOf(3)).
                                         divide(BigDecimal.valueOf(4));

    assertEquals(new BigDecimal("1.5"), newValue);
}
```




$(1 * 2 * 3) / 4 = 6/4 = 3/2 = 1.5$ (0.005 s)


Special test names




```
@DisplayName("REST product controller")
public class C_DisplayNameDemo
{
    @Test
    @DisplayName("GET 'http://localhost:8080/products/4711' user: Peter Müller")
    public void getProductFor4711()
    {
        // ..
    }

    @Test
    @DisplayName("POST 'http://localhost:8080/products/' user: Stock Manager")
    public void addProductAsStockManager()
    {
        // ...
    }
}
```

▼  REST product controller [Runner: JUnit 5] (0.007 s)

 POST 'http://localhost:8080/products/' user: Stock Manager (0.000 s)

 GET 'http://localhost:8080/products/4711' user: Peter Müller (0.007 s)



Special Assertions



Multiple Asserts



```
@Test
void multipleAssertsforOneTopic()
{
    final Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");

    // JUnit 4
    assertEquals("Mike", mike.name);
    assertEquals(LocalDate.of(1971, 2, 7), mike.dateOfBirth);
    assertEquals("Zürich", mike.homeTown);

    // JUnit 5
    assertAll(() -> assertEquals("Mike", mike.name),
              () -> assertEquals(LocalDate.of(1971, 2, 7), mike.dateOfBirth),
              () -> assertEquals("Zürich", mike.homeTown));
}
```



**Where is the
difference?**

Multiple Asserts



@Test

```
void multipleAssertsforOneTopic_Diff1() {
```

```
    Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");
```

```
    assertEquals("Tim", mike.name);
```

```
    assertEquals(LocalDate.of(1971, 3, 27), mike.dateOfBirth);
```

```
    assertEquals("Kiel", mike.homeTown);
```

```
}
```

@Test

```
void multipleAssertsforOneTopic_Diff2() {
```

```
    Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");
```

```
    assertAll((() -> assertEquals("Tim", mike.name),
```

```
                () -> assertEquals(LocalDate.of(1971, 3, 27), mike.dateOfBirth),
```

```
                () -> assertEquals("Kiel", mike.homeTown));
```

```
}
```

Multiple Asserts



@Test

```
void multipleAssertsforOneTopic_Diff1() {
```

```
    Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");
```

```
    assertEquals("Tim", mike.name);
```

```
    assertEquals(LocalDate.of(1971, 3, 27), mike.dateOfBirth);
```

```
    assertEquals("Kiel", mike.homeTown);
```

```
}
```

org.opentest4j.AssertionFailedError: expected: <Tim> but was: <Mike>
at a_first_slides.DisplayNameExample.multipleAssertsforOneTopic_Diff1(D

@Test

```
void multipleAssertsforOneTopic_Diff2() {
```

```
    Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");
```

```
    assertAll(() -> assertEquals("Tim", mike.name),
```

```
              () -> assertEquals(LocalDate.of(1971, 3, 27), mike.dateOfBirth),
```

```
              () -> assertEquals("Kiel", mike.homeTown));
```

```
}
```

org.opentest4j.MultipleFailuresError: Multiple Failures (3 failures)
expected: <Tim> but was: <Mike>
expected: <1971-03-27> but was: <1971-02-07>
expected: <Kiel> but was: <Zürich>



How do we test floating point numbers?

Special handling for floating point numbers



```
@Test
@DisplayName("\uD835\uDED1 = 3.1415 (with four digit precision)")
void floatingArithmeticRoundingForPI()
{
    double value = calculatePI();
    double precision = 0.0001;

    assertEquals(3.1415, value, precision);
}

private double calculatePI()
{
    return Math.PI;
}
```

Runs: 1/1 Errors: 0 Failures: 0

▼ DisplayNameExample [Runner: JUnit 5] (0.000 s)
 π = 3.1415 (with four digit precision) (0.000 s)



How do we test arrays?

Compare arrays - Oops!





```
@Test
void arraysCompare()
{
    final String[] words = { "Word1", "Word2" };
    final String[] expected = { "Word1", "Word2" };


    assertEquals(expected, words);
}
```

```
@Test
void nestedArraysCompare()
{
    final String[][] nested = { { "Line1", "Word1" },
                                  { "Line2", "Word2" } };
    final String[][] expected = { { "Line1", "Word1" },
                                    { "Line2", "Word2" } };

    assertEquals(expected, nested);
}
```

▼  G_ArrayEqualsWrong [Runner: JUnit 5] (0.001 s)

 arraysCompare() (0.000 s)

 nestedArraysCompare() (0.000 s)

Compare arrays **CORRECTLY**






```
@Test
void arraysCompare()
{
    final String[] words = { "Word1", "Word2" };
    final String[] expected = { "Word1", "Word2" };

    assertArrayEquals(expected, words);
}
```

```
@Test
void nestedArraysCompare()
{
    final String[][] nested = { { "Line1", "Word1" },
                                  { "Line2", "Word2" } };
    final String[][] expected = { { "Line1", "Word1" },
                                    { "Line2", "Word2" } };

    assertArrayEquals(expected, nested);
}
```

▼  G_ArrayEquals [Runner: JUnit 5] (0.000 s)
  arraysCompare() (0.000 s)
  nestedArraysCompare() (0.000 s)




How do we test Collections?



Compare Collections - no problem?!



```
@Test
void listSetCompareSameCollectionType()
{
    final Collection<String> tags = new HashSet<>(Set.of("Fast", "Cool"));
    final Collection<String> names = List.of("Tim", "Mike", "Tom");

    assertEquals(Set.of("Fast", "Cool"), tags);
    assertEquals(List.of("Tim", "Mike", "Tom"), names);
}
```

▼  ListTest [Runner: JUnit 5] (0,022 s)

-  listSetCompareSameCollectionType() (0,006 s)
-  listSetCompareCorrected() (0,010 s)

Compare Collections - What to do in case of different type







```
@Test
public void listSetCompareWrong()
{
    final List<String> actual = List.of("a", "b", "c", "d");
    final Set<String> expected = new TreeSet<>(Set.of("c", "a", "d", "b"));

    // compare set and list
    assertEquals(expected, actual);
}
```

org.opentest4j.AssertionFailedError: expected: java.util.TreeSet@3b07a0d6<[a, b, c, d]> but was:
java.util.ImmutableCollections\$ListN@11a9e7c8<[a, b, c, d]>

Runs: 3/3 ✖ Errors: 0 ✖ Failures: 1


▼  ListTest [Runner: JUnit 5] (0,010 s)
  listSetCompareSameCollectionType() (0,003 s)
  listSetCompareCorrected() (0,002 s)
  listSetCompareWrong() (0,005 s)



Compare Collections SAFER



```
@Test
public void listSetCompareCorrected()
{
    final List<String> actual = List.of("a", "b", "c", "d");
    final Set<String> expected = new TreeSet<>(Set.of("c", "a", "d", "b"));

    // compare set and list based on Iterable
    assertIterableEquals(expected, expected);
}
```

▼  ListTest [Runner: JUnit 5] (0,022 s)

-  listSetCompareSameCollectionType() (0,006 s)
-  listSetCompareCorrected() (0,010 s)



Testing Exceptions





JUnit 5: assertThrows()

```
@Test
void cannotSetValueToNull()
{
    assertThrows(NullPointerException.class,
        () -> new BigDecimal((String) null));
}
```

```
@Test
void assertThrowsException()
{
    assertThrows(IllegalArgumentException.class,
        () -> { Integer.valueOf(null); });
}
```



JUnit 5: assertThrows() with return

```
@Test
void shouldThrowExceptionAndInspectMessage()
{
    UnsupportedOperationException exception =
        assertThrows(UnsupportedOperationException.class,
            () ->
            {
                throw new UnsupportedOperationException("Not supported");
            });

    assertEquals(exception.getMessage(), "Not supported");
}
```




JUnit 5: assertThrows() with return

```
@Test
@DisplayName("Exception test clearer")
void exceptionTestImproved()
{
    Executable executable = () -> {
        throw new UnsupportedOperationException("Not supported");
    };

    UnsupportedOperationException exception =
        assertThrows(UnsupportedOperationException.class, executable);

    assertEquals(exception.getMessage(), "Not supported");
}
```



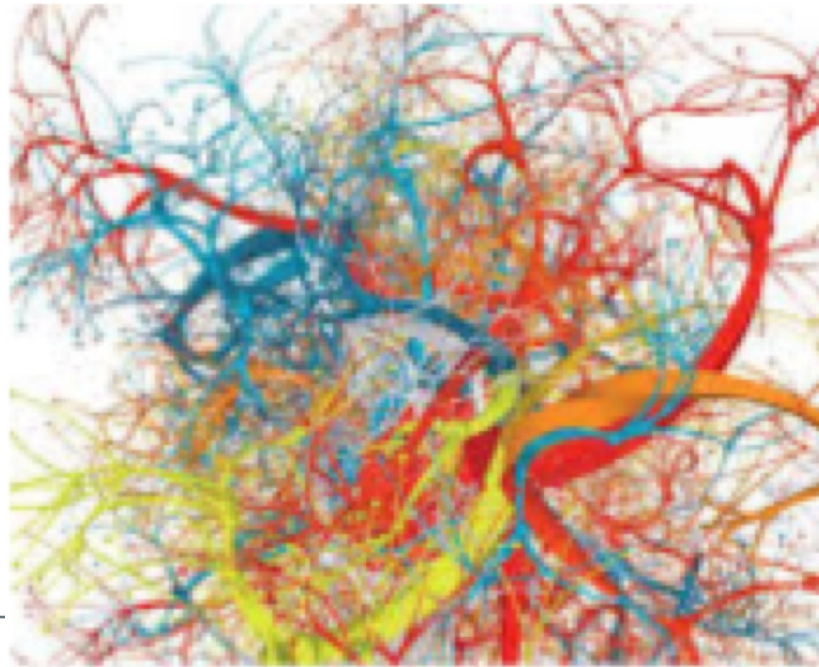
PART 3

JUnit 5 Advanced





Combinatorics / Complexity





3 key questions

1. Which value assignments should one test?
 2. How do I avoid too much effort?
 3. How do I find those test cases that allow a good and safe statement about the quality and functionality?
-

Complexity



- **Which value assignments should be tested?**
 - Even with two int int => $232 * 232 = 264$ combinations
 - **How to avoid too much overhead?**
 - Test the important / complex things
 - Do not test getters / setters
 - Clever choice of inputs so that many variants are tested
 - **How do I find those test cases that give a good and confident indication of quality and functionality?**
 - **Equivalence class test**
 - **Boundary test**
-

Equivalence classes



- Grouping of inputs: Different values => same result
- Typical example: discount calculation

| Value range | discount |
|----------------------|----------|
| count < 100 | 0 % |
| 100 <= count <= 1000 | 4 % |
| count > 1000 | 7 % |

- How many and which equivalence classes result?
-

Equivalence class test



- So let's write 3 test methods. **But: Are these tests enough?**

```
@Test
public void testCalcDiscount_SmallOrder_NoDiscount()
{
    final int smallAmount = 20;
    assertEquals(0, calculator.calcDiscount(smallAmount), "no discount");
}

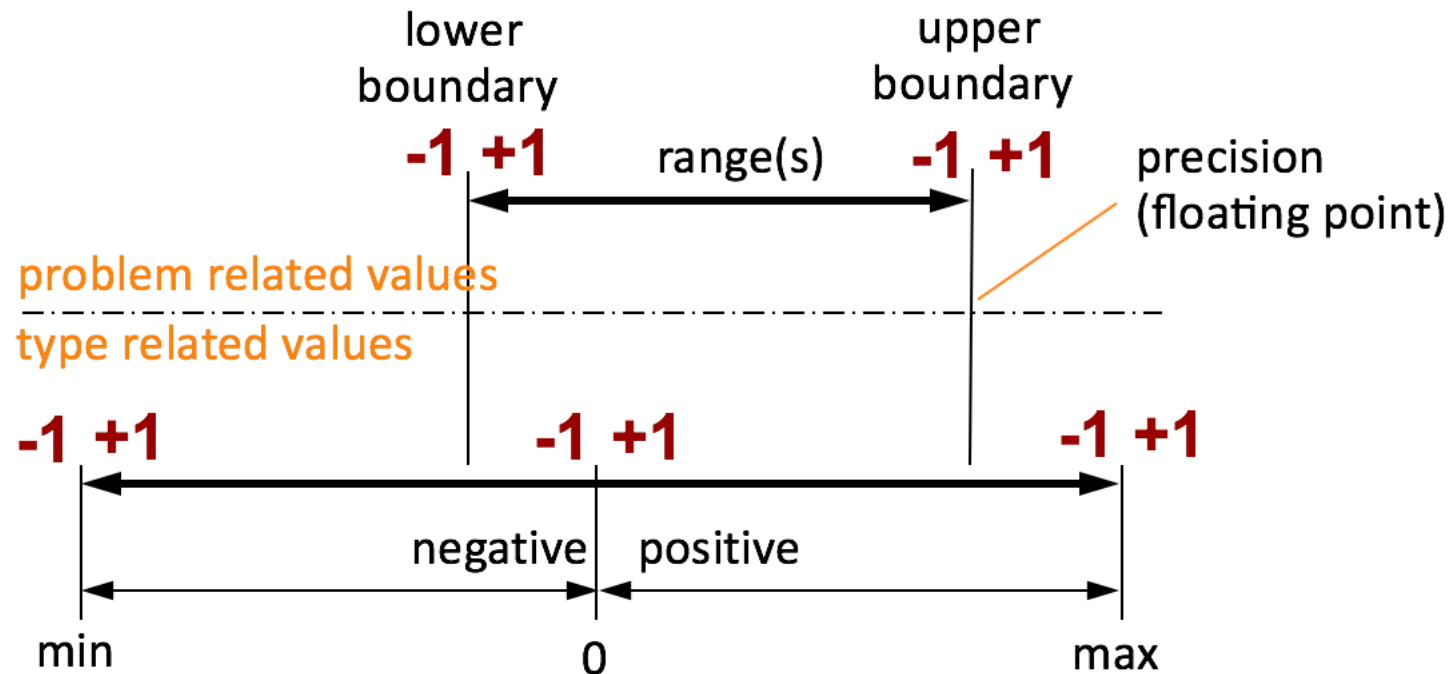
@Test
public void testCalcDiscount_MediumOrder_MediumDiscount()
{
    final int mediumAmount = 200;
    assertEquals(4, calculator.calcDiscount(mediumAmount), "4 % discount");
}

@Test
public void testCalcDiscount_BigOrder_BigDiscount()
{
    final int bigAmount = 2000;
    assertEquals(7, calculator.calcDiscount(bigAmount), "7 % discount");
}
```

Boundary tests



- **NO!** The experience from the practice shows, besides equivalence class tests one needs still others, why?
- Often we still find **problems at the edges**, i.e. in the transition of the value ranges:



Boundary tests



- For the discount calculation we still find problems at the edges, i.e. in the transition of the value ranges, here thus
 - 99, 100, 101
 - 999, 1000, 1001
- Why? Often errors in comparisons with $<$ $<=$ $=$ $!=$ $>=$ $>$
- Other potential candidates are:
 - Values < 0 or
 - Values $>$ than an intended maximum



Are we supposed to write individual methods for all these values?





Parameterized Tests





- **Remedy by so-called parameterized test**
 - **Execute test case with different data again and again with new value assignment**
 - **Thus covering all desired combinations to be tested**
 - **Realization variants**
 - Manual work: for loop: delivers only successive results
 - JUnit 4 spasmodic, syntactically unattractive
 - JUnit 4 with Expected Exception better, but again some manual work
 - JUnit 5 **finally good**
-

Parameterized Test: Short look back: for loop



```
@Test
public void testCheckMatchingBracesAllOkay() throws Exception
{
    List<String> inputs = List.of("()", "()[]{}", "[((()[]{}))]");

    for (String current : inputs)
    {
        assertTrue("Checking " + current,
                    MatchingBracesChecker.checkMatchingBraces(current));
    }
}

@Test
public void testCheckMatchingBracesAllWrong() throws Exception
{
    for (String current : List.of("(()", "({})", "({})", "())("))
    {
        assertFalse("Checking " + current,
                    MatchingBracesChecker.checkMatchingBraces(current));
    }
}
```

Parameterized Test: Short look back: JUnit 4 Parameterized



- How do we test the following class Adder with different value combinations?

```
public class Adder
{
    public int addNumbers(int a, int b)
    {
        return a + b;
    }
}
```

- So let's write a test class with
 - @RunWith(Parameterized.class)
 - A constructor and all inputs and Expected
 - A static method to generate the test data.
 - A test method








```
@RunWith(Parameterized.class)
public class AdderJUnit4Test
{
    private int first;
    private int second;
    private int expected;

    public AdderJUnit4Test(int firstNumber, int secondNumber, int expectedResult)
    {
        this.first = firstNumber;
        this.second = secondNumber;
        this.expected = expectedResult;
    }

    @Parameters(name="{0} + {1} = {2}")
    public static Collection<Integer[]> inputAndExpectedNumbers()
    {
        return Arrays.asList(new Integer[][] { { 1, 2, 3 }, { 3, -3, 0 },
                                                { 7, 2, 9 }, { 7, -2, 5 } });
    }

    @Test
    public void sum()
    {
        assertEquals(expected, Adder.add(first, second));
    }
}
```

▼  AdderJUnit4Test [Runner: JUnit 5] (0.000 s)

- ▶  [1 + 2 = 3] (0.000 s)
- ▶  [3 + -3 = 0] (0.000 s)
- ▶  [7 + 2 = 9] (0.000 s)
- ▶  [7 + -2 = 5] (0.000 s)



**That can't really be it
now?!**

Parameterized Test – JUnit 5 @ParameterizedTest / @ValueSource



```
@ParameterizedTest
@ValueSource(strings = { "Otto", "Radar" })
void palindromes(String candidate)
{
    boolean isPalindrome = PalindromeChecker.isPalindrome(candidate);

    assertTrue(isPalindrome);
}
```

▼ C_ParametrizedTest [Runner: JUnit 5] (0.005 s)

▼ palindromes(String) (0.005 s)

 [1] Otto (0.005 s)


 [2] Radar (0.006 s)


Parameterized Test – @CsvSource







```
@ParameterizedTest
@CsvSource({"Otto,true", "Radar,true", "Dummy,false" })
void palindromes2(String candidate, boolean expected)
{
    boolean isPalindrome = PalindromeChecker.isPalindrome(candidate);

    assertEquals(expected, isPalindrome);
}
```



▼  C_ParametrizedTest [Runner: JUnit 5] (0.006 s)

▼  palindromes2(String, boolean) (0.006 s)

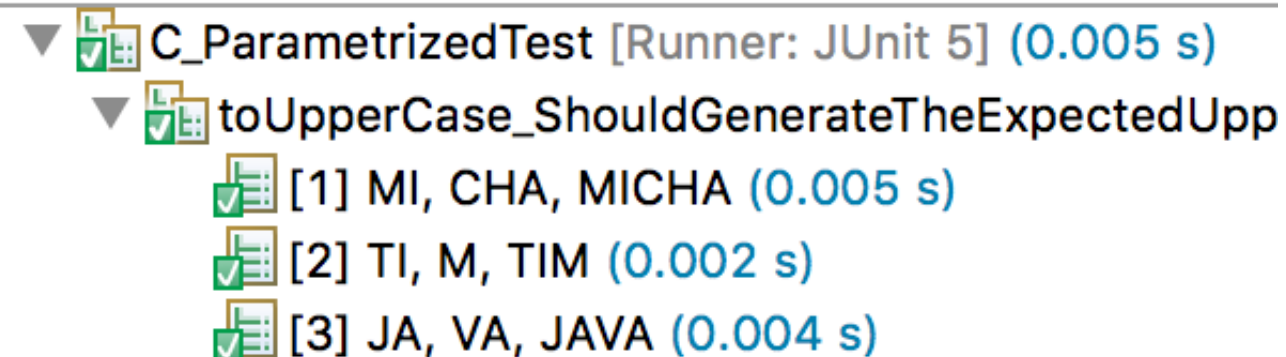
-  [1] Otto, true (0.006 s)
-  [2] Radar, true (0.002 s)
-  [3] Dummy, false (0.004 s)

Parameterized Test – multiple inputs



```
@ParameterizedTest
@CsvSource({"MI,CHA,MICHA", "TI,M,TIM", "JA,VA,JAVA"})
void toUpperCase_ShouldGenerateTheExpectedUppercaseValue(String input1,
                                                         String input2,
                                                         String expected)
{
    String actualValue = input1.concat(input2);

    assertEquals(expected, actualValue);
}
```





Parameterized Test – several inputs different types







```
@ParameterizedTest
@CsvSource({"MI,CHA,5", "TI,M,3", "JA,VA,4"})
void toUpperCase_DifferentTypes(String input1,
                                String input2,
                                int expectedLength)
{
    int actualValue = input1.concat(input2).length();

    assertEquals(expectedLength, actualValue);
}
```



▼  ParametrizedTestExample [Runner: JUnit 5] (0.000 s)

▼  toUpperCase_DifferentTypes(String, String, int) (0.000 s)







-  [1] MI, CHA, 5 (0.000 s)
-  [2] TI, M, 3 (0.000 s)
-  [3] JA, VA, 4 (0.002 s)

Parameterized Test – better naming



```
@ParameterizedTest(name = "{0} + {1} should have length {2}")
@CsvSource({"MI,CHA,5", "TI,M,3", "JA,VA,4"})
void toUpperCase_DifferentTypes_name(String input1,
                                     String input2,
                                     int expectedLength)
{
    int actualValue = input1.concat(input2).length();

    assertEquals(expectedLength, actualValue);
}
```

- 
- ▼  ParametrizedTestExample [Runner: JUnit 5] (0.009 s)
 - ▼  toUpperCase_DifferentTypes_name(String, String, int) (0.009 s)
 -  MI + CHA should have length 5 (0.009 s)
 -  TI + M should have length 3 (0.002 s)
 -  JA + VA should have length 4 (0.005 s)



**So what does the test for
the Adder look like with
JUnit 5?**

Parameterized Test – Adder Test & Better Naming



```
public class A_AdderJUnit5Test
{
    @ParameterizedTest(name = "{index}: {0} + {1} = {2}")
    @CsvSource({ "1,2,3", "3, -3, 0", "7, 2, 9", "7,-2,5" })
    void addition(int a, int b, int result)
    {
        int sum = Adder.add(a, b);

        assertEquals(result, sum);
    }
}
```

```
▼ [Success] A_AdderJUnit5Test [Runner: JUnit 5] (0.005 s)
  ▼ [Success] addition(int, int, int) (0.005 s)
    [Success] 1: 1 + 2 = 3 (0.005 s)
    [Success] 2: 3 + -3 = 0 (0.003 s)
    [Success] 3: 7 + 2 = 9 (0.002 s)
    [Success] 4: 7 + -2 = 5 (0.007 s)
```

Parameterized Test – Various conversions and help



- *LocalDate, LocalTime, LocalDateTime, Year, Month, etc.*

```
@ParameterizedTest
@ValueSource(strings = { "2019-08-01", "2019-08-31" })
void convertStringToLocalDate(LocalDate localDate)
{
    assertEquals(Month.AUGUST, localDate.getMonth());
}
```

```
@ParameterizedTest
@CsvSource(value= {"APRIL:30", "JUNE:30",
                  "JULY:31"}, delimiter = ':')
void convertStringToMonth(Month month,
                          int length)
{
    assertEquals(length,
                  month.length(false));
}
```

▼ A_ParametrizedTest_Specials [Runner: JUnit 5] (0.03 s)







- ▼ convertStringToMonth(Month, int) (0.004 s)
 - ✓ [1] APRIL, 30 (0.004 s)
 - ✓ [2] JUNE, 30 (0.002 s)
 - ✓ [3] JULY, 31 (0.007 s)
- ▼ convertStringToLocalDate(LocalDate) (0.008 s)
 - ✓ [1] 2019-08-01 (0.008 s)
 - ✓ [2] 2019-08-31 (0.007 s)

Parameterized Test – @MethodSource, e. g. for Enums



```
@ParameterizedTest
@MethodSource("createMonthsWithLength")
void withMethodSource(Month month, int expectedLength)
{
    assertEquals(expectedLength, month.length(false));
}

private static Stream<Arguments> createMonthsWithLength()
{
    return Stream.of(Arguments.of(Month.JANUARY, 31),
        Arguments.of(Month.APRIL, 30));
}
```

- ▼  A_ParametrizedTest_Specials [Runner: JUnit 5] (0.025 s)
 - ▶  convertStringToMonth(Month, int) (0.004 s)
 - ▶  convertStringToLocalDate(LocalDate) (0.007 s)
 - ▼  withMethodSource(Month, int) (0.001 s)
 -  [1] JANUARY, 31 (0.001 s)
 -  [2] APRIL, 30 (0.004 s)

Parameterized Test – @MethodSource



```
@ParameterizedTest
@MethodSource("stringIntAndListProvider")
void multipleArgumentsWithMethodSource(String str, int num, List<String> list)
{
    assertEquals(5, str.length());
    assertTrue(num < 10);
    assertEquals(3, list.size());
}

static Stream<Arguments> stringIntAndListProvider()
{
    return Stream.of(Arguments.arguments("James", 2, List.of("a", "b", "c")),
        Arguments.arguments("Peter", 7, List.of("x", "y", "z")));
}
```

Runs: 2/2 Errors: 0 Failures: 0

▼ A_ParametrizedTest_Specials [Runner: JUnit 5] (0.006 s)
 ▼ multipleArgumentsWithMethodSource(String, int, List) (0.006 s)
 [1] James, 2, [a, b, c] (0.006 s)
 [2] Peter, 7, [x, y, z] (0.012 s)

Parameterized Test – @MethodSource, e.g. for lists as parameters



```
@ParameterizedTest(name = "removeDuplicates({0}) = {1}")
@MethodSource("listInputsAndExpected")
void removeDuplicates(List<Integer> inputs, List<Integer> expected)
{
    List<Integer> result = Ex02_ListRemove.removeDuplicates(inputs);

    assertEquals(expected, result);
}
```

```
static Stream<Arguments> listInputsAndExpected()
{
    return Stream.of(Arguments.of(List.of(1, 1, 2, 3, 4, 1, 2, 3),
                                   List.of(1, 2, 3, 4)),
                  Arguments.of(List.of(1, 3, 5, 7),
                                   List.of(1, 3, 5, 7)),
                  Arguments.of(List.of(1, 1, 1, 1),
                                   List.of(1)));
}
```



**What else can
be done with a
Parameterized Test?**

Parameterized Test – @CsvSource, e.g. for large amounts of data



```
@ParameterizedTest(name = "fromRomanNumber('{1}') => {0}")
@CsvSource({ "1, I", "2, II", "3, III", "4, IV", "5, V", "7, VII", "9, IX",
            "17, XVII", "40, XL", "90, XC", "400, CD", "444, CDXLIV", "500, D",
            "900, CM", "1000, M", "1666, MDCLXVI", "1971, MCMLXXI",
            "2018, MMXVIII", "2019, MMXIX", "2020, MMXX", "3000, MMM"})
@DisplayName("Convert roman number to arabic number")
void fromRomanNumber(final int arabicNumber, final String romanNumber)
{
    int result = RomanNumbers.fromRomanNumber(romanNumber);

    assertEquals(arabicNumber, result);
}
```

Parameterized Test – @CsvSource, e.g. for large amounts of data



```
@ParameterizedTest(name = "fromRomanNumber('{1}') => {0}")
@CsvFileSource(resources = "arabicroman.csv", numLinesToSkip = 1)
@DisplayName("Convert roman number to arabic number ")
void fromRomanNumber(final int arabicNumber, final String romanNumber)
{
    int result = RomanNumbers.fromRomanNumber(romanNumber);

    assertEquals(arabicNumber, result);
}
```

```
arabic,roman
1, I
2, II
3, III
4, IV
5, V
7, VII
9, IX
17, XVII
40, XL
90, XC
```

Parameterized Test – Check edge cases



```
@ParameterizedTest(name = "test '{0}' for null or empty")
@NullSource
@EmptySource
void nullAndEmptyStrings(String str)
{
    assertTrue(str == null || str.isEmpty());
}
```

```
@ParameterizedTest(name = "test '{0}' for null or empty")
@NullSource
@EmptySource
void nullAndEmptyLists(List<?> list)
{
    assertTrue(list == null || list.isEmpty());
}
```

```
▼ A_ParameterizedNullValuesTest [Runner: JUnit 5] (0,002 s)
  ▼ nullAndEmptyStrings(String) (0,025 s)
    ✓ test 'null' for null or empty (0,025 s)
    ✓ test "" for null or empty (0,004 s)
  ▼ nullAndEmptyLists(List) (0,002 s)
    ✓ test 'null' for null or empty (0,002 s)
    ✓ test '[]' for null or empty (0,004 s)
```



PART 4:

Test Smells



Test Smells High Level



- **Advice: Follow your nose 😊**
 - Basically tests are also just source code like business code
 - Test Smells are similar to Bad Smells / Code Smells (details in "Java-Profi")
 - Bad Smell Examples: https://www.youtube.com/watch?v=9E6_zpx3q2c
 - You can also find the following test smells on high level
 - Tests are difficult to write
 - Strange, complicated things needed to write / run tests
 - It needs a lot of mocking* (or even worse PowerMock for private / static mocking)
 - Testing sometimes even requires special handling in business code
 - Test execution is (too) slow
 - Test execution gives fluctuating results (random failures)
-

| Test Smell | Symptom |
|--------------------------|---|
| Hard-to-Test Code | Code is difficult to test |
| Fragile Test | A test fails to compile or run when the SUT is changed in ways that do not affect the part the test is exercising. |
| Erratic Test | One or more tests are behaving erratically; sometimes they pass and sometimes they fail. |
| Obscure Test | It is difficult to understand the test at a glance |
| Assertion Roulette | It is hard to tell which of several assertions within the same test method caused a test failure. |
| Slow Tests | The tests take too long to run. |
| | |
| Test Code Duplication | The same test code is repeated many times => Does NOT apply to simple initializations. |
| Test Logic in Production | The code that is put into production contains logic that should be exercised only during tests. => Does NOT apply to getter!* Anecdote Reflection !!!! |

Assertion Roulette / Obscure Test – Not SRP – Multiple Test Cases



```
@Test
public void testFlightMileage_asKm2() throws Exception
{
    // setup fixture
    String validFlightNumber = "LX 857";
    // exercise constructor
    Flight newFlight = new Flight(validFlightNumber);
    // verify constructed object
    assertEquals(validFlightNumber, newFlight.number);
    assertEquals("LX", newFlight.airlineCode);
    // setup mileage
    newFlight.setMileage(1111);
    int actualKilometres = newFlight.getMileageAsKm();
    // verify results
    int expectedKilometres = 1777;
    assertEquals(expectedKilometres, actualKilometres);
    // now try it with a canceled flight:
    newFlight.cancel();
    Throwable th = assertThrows(InvalidRequestException.class,
                                () -> newFlight.getMileageAsKm());
    assertEquals("Cannot get cancelled flight mileage", th.getMessage());
}
```

(vased on <http://xunitpatterns.com/Assertion%20Roulette.html>)

Test Smell: Wrong use of assertTrue()/False()



```
assertTrue(db.writeCount == 10);  
assertTrue(tasks.totalProcessed == 10);  
assertTrue(tasks.errorCount == 0);
```

Test Smell: Wrong use of assertTrue()/False()



```
assertTrue(db.writeCount == 10);  
assertTrue(tasks.totalProcessed == 10);  
assertTrue(tasks.errorCount == 0);
```



```
assertEquals(10, db.writeCount);  
assertEquals(10, tasks.totalProcessed);  
assertEquals(0, tasks.errorCount);
```

Test Smell: ☹️ assertTrue() forever? ☹️



```
@Test
void assertTrueForever()
{
    final Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");
    final Person sameMike = mike;
    final Person otherMike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");

    assertTrue(mike != null, "mike not null");
    assertTrue(mike == sameMike, "same obj");
    assertTrue(mike.equals(otherMike), "same content");
}
```

Test Smell: ☹️ assertTrue() forever? ☹️



```
@Test
void assertTrueForever()
{
    final Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");
    final Person sameMike = mike;
    final Person otherMike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");

    assertTrue(mike != null, "mike not null");
    assertTrue(mike == sameMike, "same obj");
    assertTrue(mike.equals(otherMike), "same content");
}
```



```
@Test
void rightAssertsForTheJob()
{
    final Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");
    final Person sameMike = mike;
    final Person otherMike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");

    assertNotNull(mike, "mike not null");
    assertSame(mike, sameMike, "same obj");
    assertEquals(mike, otherMike, "same content");
}
```

Test Smell: Too many Asserts



```
assertEquals(10, db.readCount);  
assertEquals(10, db.writeCount);  
assertEquals(10, db.commitCount);  
  
assertEquals(10, tasks.totalProcessed);  
assertEquals(0, tasks.errorCount);  
assertEquals(SUCCESS, tasks.status);
```


Test Smell: Use of toString() in assertEquals()



```
assertEquals("mongodb.writeConcern.timeout=10000, " +  
    "mongodb.writeConcern.writes=1, " +  
    "mongodb.port=27017, " +  
    "mongodb.password=ksdj2455aAYdsj, " +  
    "mongodb.user=ABCD",  
    dbConnection.toString())
```

Test Smell: Conditional Logic



```
@Test
void badConditionalLogic()
{
    final Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");
    final Person otherMike = new Person("Mike", LocalDate.of(1971, 2, 7), "Kiel");

    assertNotNull(mike, "mike not null");
    if (mike.getHomeTown().equals("Zürich"))
    {
        assertEquals(LocalDate.of(1971, 2, 7), mike.getDateOfBirth());
    }
    else
    {
        assertTrue(mike.equals(otherMike), "same content");
    }
}
```

Test Smell: Over Asserting



```
@Test
void overAssertingAndLoosingHelpfulInfo()
{
    List<String> result = calcResultList();
    assertEquals(1, result.size());
    assertEquals("Tom", result.get(0)); // Will not be executed
}

private List<String> calcResultList()
{
    return List.of("Tom", "Jerry");
}
```

well meant != well done

org.opentest4j.AssertionFailedError: expected: <1> but was: <2>

Test Smell: Over Asserting



```
@Test
```

```
void overAssertingAndLoosingHelpfulInfo()
```

```
{  
    List<String> result = calcResultList();
```

```
    assertEquals(1, result.size());
```

```
    assertEquals("Tom", result.get(0)); // Will not be executed
```

```
}
```

! org.opentest4j.AssertionFailedError: expected: <1> but was: <2>



```
@Test
```

```
void reasonableAssertProvidingGoodFeedback()
```

```
{  
    List<String> result = calcResultList();
```

```
    assertEquals(List.of("Tom"), result);
```

```
}
```

! org.opentest4j.AssertionFailedError: expected: <[Tom]> but was: <[Tom, Jerry]>



Part 5:

Characterization Testing

aka Pinning Test / Pin-Down Tests



Characterization Testing



- Legacy code ... big chunk of code but no tests!
- How to pin down / nail down the behaviour of thos method?

```
public static String formatText(String text)
{
    StringBuffer result = new StringBuffer();
    for (int n = 0; n < text.length(); ++n) {
        int c = text.charAt(n);
        if (c == '<') {
            while(n < text.length() && text.charAt(n) != '/' && text.charAt(n) != '>')
                n++;
            if (n < text.length() && text.charAt(n) == '/')
                n+=4;
            else
                n++;
        }
        if (n < text.length())
            result.append(text.charAt(n));
    }
    return new String(result);
}
```

Characterization Testing



- What about even larger chunks of code?

```
public class TravelsAdapter {
    public List<Travel> adapt(JsonNode jsonNode) throws InvalidTravelException {
        List<Travel> travels = new ArrayList<>();
        JsonNode payloadNode = jsonNode.with("data");
        if (payloadNode.findValue("orderId") == null ||
            StringUtils.isBlank(payloadNode.findValue("orderId").textValue())) {
            throw new InvalidTravelException("Invalid order id");
        }
        long orderId = payloadNode.findValue("orderId").asLong();
        JsonNode flights = payloadNode.withArray("flights");
        if (flights.size() == 0) {
            throw new InvalidTravelException("Invalid json (no flights)");
        }
        flights.iterator().forEachRemaining(flight -> {
            ObjectNode nodeFlight = (ObjectNode) flight;
            if (nodeFlight.get("flightId") == null || StringUtils.isBlank(nodeFlight.get("flightId").textValue())) {
                try {
                    throw new InvalidTravelException("Invalid flightNumber value");
                } catch (InvalidTravelException e) {
                    e.printStackTrace();
                }
            }
            String flightNumber = nodeFlight.get("flightId").textValue();
            String arrivalAirport = nodeFlight.get("to").textValue();
            String departureAirport = nodeFlight.get("from").textValue();
            String airline = nodeFlight.get("airline").textValue();
            travels.add(new Travel(
                orderId,
                flight.toString(),
                flightNumber,
                airline,
                departureAirport,
                arrivalAirport));
        });
        return travels;
    }
}
```




**How to describe the
behaviour if you just
have the source code?**



**Just execute with
different inputs and
observe the results!**

Characterization Testing



- The purpose of characterization testing is to document your system's actual behavior, not check for the behavior you wish your system had.
- More concrete:
 - Create test methods
 - Step 1: "silly result"

@Test

```
public void formatsPlainText1() {  
    assertEquals("XXXX", UnknownFormatterFunctionality.formatText("plain text"));  
}
```

```
public static String formatText(String text)  
{  
    StringBuffer result = new StringBuffer();  
    for (int n = 0; n < text.length(); ++n) {  
        int c = text.charAt(n);  
        if (c == '<') {  
            while(n < text.length() && text.charAt(n) != '/' && text.charAt(n) != '>')  
                n++;  
            if (n < text.length() && text.charAt(n) == '/')  
                n+=4;  
            else  
                n++;  
        }  
        if (n < text.length())  
            result.append(text.charAt(n));  
    }  
    return new String(result);  
}
```

Characterization Testing



- Just execute with different inputs and observe the result
- More concrete:
 - Create test methods
 - Step 1: "silly result"
 - Step 2: adjust "silly result" to what is delivered by method

@Test

```
public void formatsPlainText2() {  
    assertEquals("plain text", UnknownFormatterFunctionality.formatText("plain text"));  
}
```

```
public static String formatText(String text)  
{  
    StringBuffer result = new StringBuffer();  
    for (int n = 0; n < text.length(); ++n) {  
        int c = text.charAt(n);  
        if (c == '<') {  
            while(n < text.length() && text.charAt(n) != '/' && text.charAt(n) != '>')  
                n++;  
            if (n < text.length() && text.charAt(n) == '/')  
                n+=4;  
            else  
                n++;  
        }  
        if (n < text.length())  
            result.append(text.charAt(n));  
    }  
    return new String(result);  
}
```

Characterization Testing



- Just execute with different inputs and observe the result
- More concrete:
 - Create test methods
 - Step 1: "silly result»
 - Step 2: adjust "silly result" to what is delivered by method
 - Step 3: adjust test name to gain better understanding

```
@Test
public void doesNotChangeUntaggedText() {
    assertEquals("plain text", UnknownFormatterFunctionality.formatText("plain text"));
}
```

```
public static String formatText(String text)
{
    StringBuffer result = new StringBuffer();
    for (int n = 0; n < text.length(); ++n) {
        int c = text.charAt(n);
        if (c == '<') {
            while(n < text.length() && text.charAt(n) != '/' && text.charAt(n) != '>')
                n++;
            if (n < text.length() && text.charAt(n) == '/')
                n+=4;
            else
                n++;
        }
        if (n < text.length())
            result.append(text.charAt(n));
    }
    return new String(result);
}
```

Characterization Testing ... other tries



```
@Test
public void emptyInput() {
    assertEquals("XXXX", UnknownFormatterFunctionality.formatText("<>"));
}
```

org.opentest4j.AssertionFailedError: expected: <XXXX> but was: <>

```
@Test
public void removesTagTextBetweenAngleBracketPairs() {
    assertEquals("XXXX", UnknownFormatterFunctionality.formatText("<TAGGED>"));
}
```

org.opentest4j.AssertionFailedError: expected: <XXXX> but was: <>

```
@Test
public void justKeepsContent() {
    assertEquals("XXXX", UnknownFormatterFunctionality.formatText("<TAG>CONTENT<TAG>"));
}
```

org.opentest4j.AssertionFailedError: expected: <XXXX> but was: <CONTENT>

Characterization Testing ... adjusted



```
@Test
public void emptyInput2() {
    assertEquals("", UnknownFormatterFunctionality.formatText("<>"));
}

@Test
public void removesTagTextBetweenAngleBracketPairs2() {
    assertEquals("", UnknownFormatterFunctionality.formatText("<TAGGED>"));
}

@Test
public void justKeepsContent2() {
    assertEquals("CONTENT",
        UnknownFormatterFunctionality.formatText("<TAG>CONTENT<TAG>"));
}
```



Demo

Characterization Testing



- <https://michaelfeathers.silvrback.com/characterization-testing>
 - <https://www.artima.com/weblogs/viewpost.jsp?thread=198296>
 - <https://daedtech.com/characterization-tests/>
 - <https://www.fabrizioduroni.it/2018/03/20/golden-master-test-characterization-test-legacy-code/>
 - <https://freol35241.medium.com/pinning-tests-in-python-using-pytest-c678a2a3cb3b>
-



Questions?



Further info / sources



- **JUnit 5**

- <https://junit.org/junit5/>
- <https://jaxenter.de/highlights-junit-5-65986>
- <https://jaxenter.de/junit-5-beyond-testing-framework-81787>
- https://gul.gu.se/public/pp/public_courses/course82759/published/1524658283418/resourceId/40520654/content/junit-tdd-mocking.pdf
- https://www.viadee.de/wp-content/uploads/JUnit5_javaspektrum.pdf

- **AssertJ**

- <https://assertj.github.io/doc/>
 - <https://joel-costigliola.github.io/assertj/assertj-core-quick-start.html>
 - <https://www.vogella.com/tutorials/AssertJ/article.html>
 - <https://dzone.com/articles/assertj-and-collections-introduction>
 - <https://de.slideshare.net/tsveronese/assert-j-techtalk> (Hamcrest vs. AssertJ)
-



Thank You
