Junit 5-Testing-Workshop Exercises

Michael Inden

E-**Mail:** michael_inden@hotmail.com

PART I: Introduction Testing — 15 — 30 min

Exercise 1: Brainstorming

Answer the question: What makes it difficult for us to write tests?

Exercise 2: Installation of MoreUnit

Install the plugin MoreUnit via the Eclipse Marketplace or from the page

`http://moreunit.sourceforge.net/update-site/`

or for IntelliJ under

`https://plugins.jetbrains.com/plugin/7105-moreunit`

Exercise 3: Installation of Sample Project

Install the sample programs and exercises on your computer.

PART II: JUnit 5 Intro — 45 — 60 min

Exercise 1: Calculator

Exercise 1a: Create a simple class `Ex01_Calculator` and a method `add()` to add and `divide()` to divide two numbers.

Exercise 1b: Create corresponding test methods:
a) for any two positive numbers
b) the same value positive and negative: how to prettify the naming?

Exercise 1c: For exceptions
a) a division by 0.
b) Check the text of the exception

Exercise 1d: Extend the calculator with the following method and write a test that has a meaningful name, like 0.1 + 0.1 + 0.1 = 0.3

```
public double sumUp_0_1(final int factor)
{
	double value = 0.1;
	double result = 0;
	for (int i = 0; i < factor; i++)
	{
		result += value;
	}
	return result;
}
```

Exercise 2: Long Runner

Exercise 2a: Look at the `Ex02_LongRunner` class and the associated test, then run it. Why does it take so long? Improve the situation, but change ONLY in the test.

Exercise 2b: iHow to check if the result is correct even if there is a timeout? Call the method `calcFib30()` in a test and choose 1 second as timeout.

Tip: **Remember `assertThrows()` and how to get the result.**

Exercise 3: Person Creation

Look at the class `Ex03_PersonWithAddress` and the corresponding test. Then execute it. Fix the errors! Improve both in the test and in the Java code. How do you get all the errors right away?

Exercise 4: Test Inspection

Extract the following info given as annotations and check them with appropriate assert calls.

```
class Ex04_TestInfoExampleTest
{
    @Test
    @Tag("Fast")
    @Tag("Cool")
    @DisplayName("5 + (-5) => 0 🤔")
    void mytestmethod(TestInfo ti)
    {
      // TODO
    }
 }
```

Exercise 5: Arrays and Collections

Exercise 5a: The utility class `Arrays` as well as `Collections` provide methods for sorting data structures. Improve and simplify the existing tests in the `Ex05_ArraysSortTest` class that sort some values and then test the sorted result.

Exercise 5b: Simplify the test for `listRemoveDuplicates()` using JUnit 5 facilities.

Exercise 6: Test Order

Exercise 6a: Analyze the given class `Ex06_LRUCache`, which provides a Last Recently User Cache based on a `LinkedHashMap<K,V>`. Add appropriate calls to `assert` methods to correctly test the functionality `testIntersectWithGets()`.

Exercise 6b: Simplify the tests with JUnit 5 means, such as `@Order`.

PART III/IV: JUnit 5 Advanced / Migration — 60 — 75 min

Exercise 1: String Reverse

Given a class `Ex01_StringUtils` that provides a method `reverse(String)` that reverses the letters in a string. Verify this method with appropriate tests.

Tip: **`@ParameterizedTest, @CsvSource`**

Exercise 2: Well Formed Braces

Given is an implementation `Ex02_MatchingBracesChecker`, which checks whether braces are well-formed, that is, always an opening and then a matching closing brace occurs, for example for

```
String input1 = "()[]{}";
String input2 = "[((()[]{}))]";
String inputWrong1 = "(()";
String inputWrong2 = "(()})";
```

Exercise 2a: Write appropriate tests and try to find and correct two bugs in the implementation. Extend the tests if necessary.

Exercise 2b: Improve the implementation of tests using Parameterized Tests.

Exercise 3: Leap Year: Conversion to Parameterized Test with hint

Given a calculation of leap years in a class `Ex03_LeapYear`. Improve the implementation of the tests in the class `Ex03_LeapYearTest`.

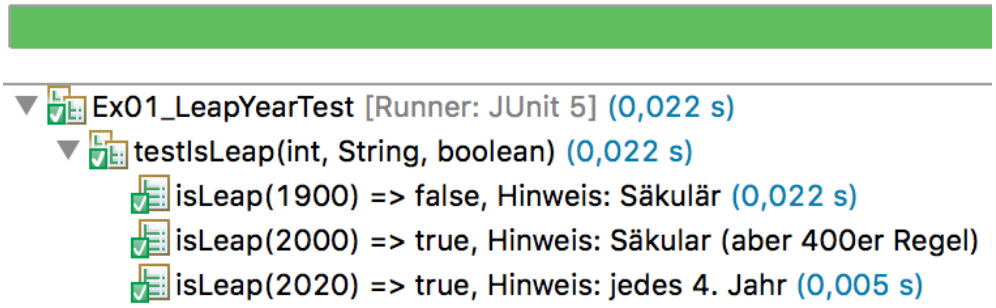
There are a few special cases as well as the rule of 4:

- If a year is divisible by 4, it is usually a leap year.
- Years that are divisible by 100 are called secular years and are not leap years.
- However, secular years that are also divisible by 400 are leap years again.

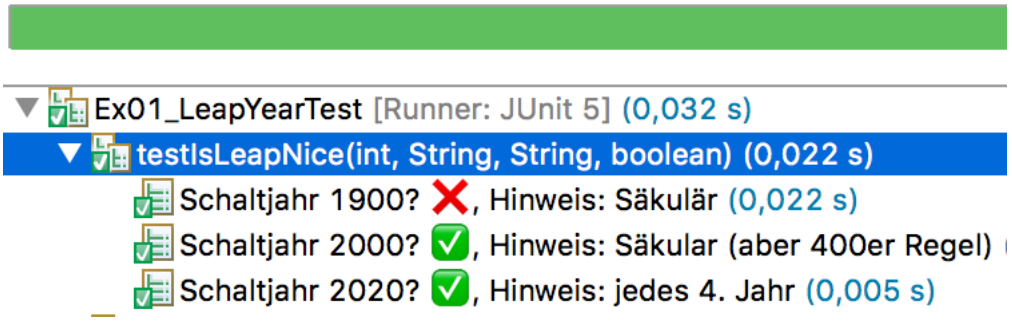
Exercise 3a: These rules can be tested individually as follows, but since JUnit 5 a parametrized test is the obvious choice. Convert them into one, starting with `@ValueSource(ints):`

```
@Test
void testIsLeap_4_Years_Rule()
{
      final boolean result = Ex01_LeapYear.isLeap(2020);

      assertTrue(result);
}
```

Exercise 3b: There should not only be a simple test, but also an indication of why such a decision was made. For this purpose, write a test method that produces output such as the following:



Exercise 3c: Improve readability as follows:



Aufgabe 4: addOne

Given an array of numbers representing the digits of a number, such as `[1, 2, 3, 4]` for the value 1234. The method `int[] addOne(int[])` adds the value 1 to this number, thus producing the following result `[1, 2, 3, 5]` for the example.

Exercise 4a: The original developer created the following unit test and quickly left because it was **RED**. Analyze and fix the problem.

```
@Test
@DisplayName("[1, 2, 3, 4] + 1 => [1, 2, 3, 5]")
void testAddOne()
{
	final int[] values = { 1, 2, 3, 4 };
	final int[] expected = { 1, 2, 3, 5 };

	final int[] result = Ex04_AddOneToAnArray.addOne(values);

	assertEquals(expected, result);
}
```

Exercise 4b: Now that a first test exists as a basis, further test cases should be added. Consider which special features the addition possesses and which test cases should be

Tip: Use `@MethodSource`.

Exercise 5: Add Roman Numbers

Given a class `Ex05_RomanNumbers` for conversion to and from Roman numbers with the two methods:

* `int fromRomanNumber(String)`
* `String toRomanNumber(int)`

Conveniently, both are already well tested with unit tests.

But now the business comes and wants to provide an arithmetic addition functionality. Develop this in a class `Ex05_RomanNumberAdder` and rely on unit tests. The business is willing to contribute a CSV file `roman-addition.csv`, which looks shortened as follows:

```
Roman     Roman     Sum       Calculation = Value
-----------------------------------------------
I,        I,        II,       1 + 1 = 2
I,        II,       III,      1 + 2 = 3
I,        III,      IV,       1 + 3 = 4
I,        IV,       V,        1 + 4 = 5
V,        II,       VII,      5 + 2 = 7
V,        IV,       IX,       5 + 4 = 9
X,        VII,      XVII,     10 + 7 = 17
X,        XX,       XXX,      10 + 20 = 30
```

Exercise 6: Payment Day with note

Given a calculation of the next salary payday based on a `LocalDate`. The following rules apply:

1) The salary is paid on the 25th of the month. In December, it is paid in the middle of the month.
2) If the payday falls on a weekend, the Friday before is the payday. Still, in December, it is the following Monday.

For this purpose a class `NextPaydayAdjuster` was developed. **Think about the necessary test cases and create appropriate unit tests with an understandable hint**, such as "Friday if 25th on the weekend".

Tip 1: Correct the implementation if errors are discovered through test cases.

Tip 2: Use a `@CsvSource` and modify the separator.

Exercise 7: Discount Calculation

We want to determine the discount for an item based on the quantity of the order. The following requirement from the business is given:

Wertebereich	Rabatt
count < 50	0 %
50 <= count <= 1000	4 %
count > 1000	7 %

In addition, a diligent developer has already provided the following implementation:

```
public class Ex07_DiscountCalculator
{
	// ATTENTION: Deliberately contains a few small errors
	public int calcDiscount(final int count)
	{
		if (count < 50)
			return 0;
		if (count > 50 && count < 1000)
			return 4;
		if (count > 1000)
			return 7;

		throw new IllegalStateException("programming " +
			"problem: should never reach this line.“ +
			"value " + count + " is not handled!");
	}
}
```

Exercise 7a: Check the implementation of equivalence class tests and simplify them using Parameterized Tests.

Exercise 7b: Add the boundary value tests given as examples. This should allow you to detect errors at edges. Correct the implementation if necessary. Simplify the boundary tests by using Parameterized Tests.

Exercise 7c: Add a check for positive values and secure this with an appropriate test.

Exercise 7d: Vary the return so that instead of a number, it returns values from the `Discount` enum. Create the class `Ex07_DiscountCalculator_WithEnum`.
Modify the tests so that enum values are used for parameterized tests.

Tip: **`@MethodSource`**

Exercise 8: Argument Converter

Sometimes the conversions provided by JUnit 5 are not sufficient. However, it is possible to create your own converters in a simple way as a workaround.

Exercise 8a: Write a simple HexConverter of your own, which allows to specify hexadecimal numbers for the following test - if you like binary numbers, add the second converter.

```
@ParameterizedTest
@CsvSource({ "F, 15", "10, 16", "AA, 170", "FF, 255" })
void hexConverter(@ConvertWith(HexToInt.class) int input,
                  int expected)
{
      assertEquals(expected, input);
}

@ParameterizedTest
@CsvSource({ "1, 1", "10, 2", "111, 7", "11111111, 255" })
void binaryConverter(@ConvertWith(BinaryToInt.class) int input,
                     int expected)
{
      assertEquals(expected, input);
}
```

Exercise 8b: Write an ArgumentConverter that transforms a string representation of an array or list, that is `[ Value1, Value2, Value3 ]` into a list of desired values, in the following example from the string-based date values into a `List<LocalDate>`:

```
@ParameterizedTest(name = "sundays between {0} and {1} => {2}")
@MethodSource("startAndEndDateAndArrayResults")
void sundaysBetween(LocalDate start, LocalDate end,
                    @ConvertWith(FromStringArrayConverter.class)
List<LocalDate> expected)
{
    final List<LocalDate> result =
          SundayCalculator.allSundaysBetween(start, end);

    assertEquals(expected, result);
}

private static Stream<Arguments> startAndEndDateAndArrayResults()
{
    return Stream.of(Arguments.of(LocalDate.of(2020, 1, 1),
                                  LocalDate.of(2020, 3, 1),
          "[2020-01-05, 2020-01-12, 2020-01-19, 2020-01-26," +
          " 2020-02-02, 2020-02-09, 2020-02-16,2020-02-23]"));
}
```

Exercise 9: JSON Argument Converter

Sometimes the conversions provided by JUnit 5 are not sufficient. However, it is easy to create your own converters as a workaround. In this task, a JSON converter is to be created.

Tip: Use the external library GSON.
https://mvnrepository.com/artifact/com.google.code.gson/gson

```
@ParameterizedTest
@CsvSource(value = {
"{ name:'Peter', dateOfBirth: '2012-12-06', homeTown : 'Köln'} |
false",
"{ name:'Mike', dateOfBirth: '1971-02-07', homeTown : 'Zürich'} | true"
}, delimiter = '|')
void jsonPersonAdultTest(@ConvertWith(JsonToPerson.class)
                         Person person, boolean expected)
{
      final long age = ChronoUnit.YEARS.between(person.dateOfBirth,
                                                LocalDate.now());

      assertEquals(expected, age >= 18);
}

static class JsonToPerson extends SimpleArgumentConverter
{
      @Override
      protected Person convert(Object source, Class<?> targetType)
      {
            // TODO
            return null;
      }
}
```

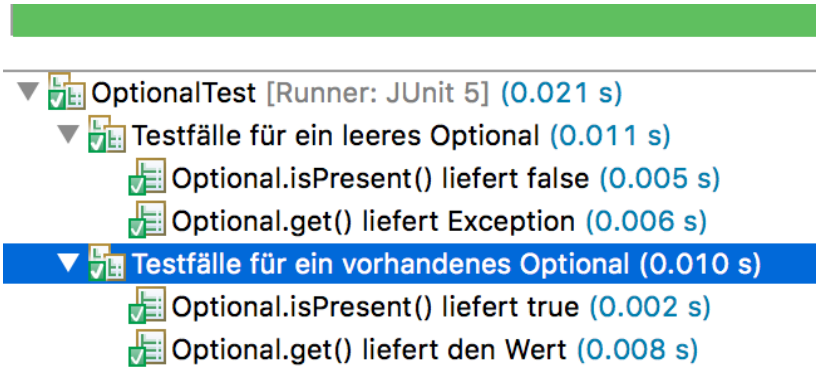
Exercise 10: Benchmark Extension

Create an extension to determine the running times of the individual test cases. For example, use calls to `FibonacciCalculator.fibRec(47)` to provoke longer runtimes. For another test case, limit the runtime to a maximum of 2 seconds.

Tip: **`@ExtendWith(BenchmarkExtension.class)`**

Exercise 11: Nested Test

Write a test that checks the essential functionalities of the `Optional<T>` class. (https://www.viadee.de/wp-content/uploads/JUnit5_javaspektrum.pdf).



Exercise 12: AssertJ

Given a list of people and a set of comparators.

```
private static final List<Person> persons =
                                  new ArrayList<>(List.of(
 new Person("Mike",  LocalDate.of(1971,  2,  7), "Bremen"),
 …
 new Person("Tom",   LocalDate.of(2011, 11, 11), "Aachen")));

private final Comparator<Person> byName =
              Comparator.comparing(Person::getName);
private final Comparator<Person> byBirthday =
              Comparator.comparing(Person::getDateOfBirth);
```

To test the respective sorting, three test cases exist, each defining the desired results.

Exercise 12a: How can AssertJ make the whole process much shorter and easier?

Exercise 12b: What is the big advantage about the verification implemented with AssertJ.

Exercise 13: Permutations

In the class `Ex09_StringUtils` you can find a method `alcPermutations(String)`, which determines all permutations for a given text, e.g.:

A	=>	A
AB	=>	AB, BA
ABC	=>	ABC, ACB, BAC, BCA, CAB, CBA

Exercise 13a: Write unit tests for these three cases.

Exercise 13b: Simplify this with a parameterized test. Consider another form of parameter provisioning here. (@MethodSource) (`@MethodSource`)

Exercise 13c: Quickly the result set becomes quite large, because the number of factorials corresponds to the length of the string. How do you proceed with the following data? How to check all values? Is it possible at all? What are the alternatives?

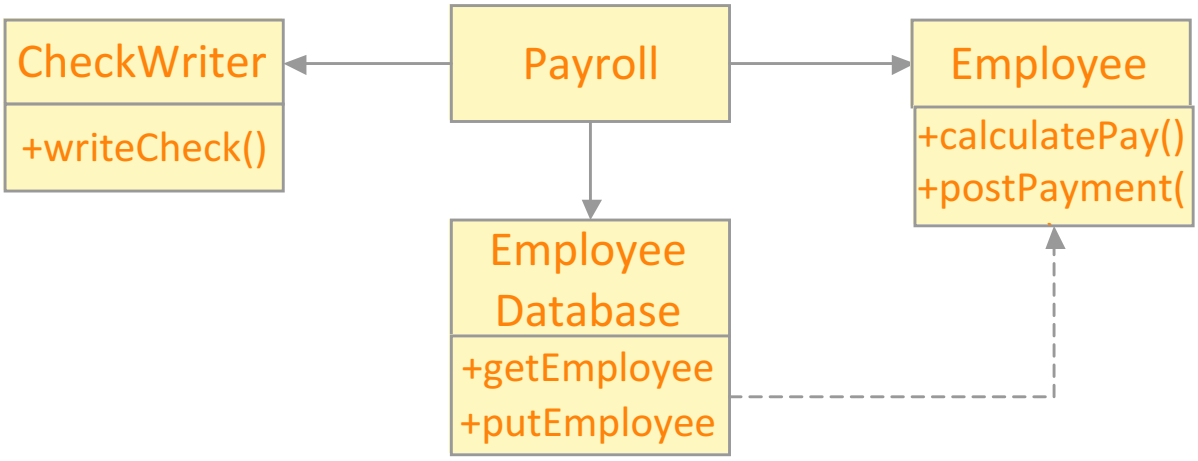
```
@Test
void testManyPermutations()
{
    final String input = "0123456789";
    final int expectedSizeBasedOnFaculty =
              MathUtils.fac(input.length());

    final Set<String> permutations =
                      Ex03_StringUtils.calcPermutations(input);

    assertEquals(expectedSizeBasedOnFaculty, permutations.size());
    // and now??? how to check all the values??? Ideas?
}
```

Exercise 13d: So far, we have completely disregarded a rather important special case in the tests, namely that the strings can also repeat and thus the number of combinations can be drastically reduced. In fact, then the following applies:
number of combinations = n! / k!, where n is the length and k the number of duplicates. Write a suitable test for the input "AACAA" with the given auxiliary methods, `countDuplicates(String)` as well as `fac(int)` and the information about the number of combinations.

PART V/VI: Test modes and dependencies / Design For Testability — 60 — 75 min

Exercise 1: Target Breaking Points — Design for Testability

Given the following class diagram that still contains some direct dependencies. How and where do you have to change the design to make the `Payroll` class more testable:
Modify the design so that the `Payroll` class becomes testable independently of the database and other classes. (Example is taken from "Agile Software Development" by Robert C. Martin)



Exercise 2: Extract And Override

Given the following class `PizzaService`.

```
public class PizzaService
{
    private final SmsNotificationService notificationService;

    public PizzaService()
    {
        notificationService = new SmsNotificationService();
    }

    public void orderPizza(final String name)
    {
        notificationService.send("Pizza " + name +
                                 " wird in Kürze geliefert.");
    }
}
```

However, due to the nature of its implementation, it is not particularly testable: it sends an SMS notification when orders are placed - simplistically simulating the sending of SMS by presenting a dialog:

If we wanted to test this construct, this is quite problematic:

1) Surely an SMS should not be sent on every test run.
2) Also, presenting a dialog hinders the automation of tests.

Let's try to write a unit test anyway and remember the ARRANGE-ACT-ASSERT style:

```
public class PizzaServiceTest
{
    @Test
    public void orderPizza_should_send_sms()
    {
        // Arrange
        final PizzaService service = new PizzaService();

        // Act
        service.orderPizza("Diavolo");
        service.orderPizza("Surprise");

        // Assert ???
    }
}
```

The test shows several things: With state-based testing (that is, querying data), we cannot check whether the messages have been sent or not. Moreover, the negative points mentioned above still exist. So what can we do? We need to ...

1) introduce a breaking point in the `PizzaService` class,
2) create a stub implementation for the `SMSNotificationService` and
3) make some adjustments in the unit test.

Exercise 3: Mockito First Steps

Given is a simple test class `Ex03_MockitoBasicsTest`, which has to be extended at the respective places, so that the test cases are passed successfully.

```
    @Test
    public void iterator_next_return_hello_world()
    {
        Iterator<String> it = Mockito.mock(Iterator.class);
        // TODO

        String result = it.next() + " " + it.next();

        assertEquals("Hello World", result);
    }
```

Exercise 4: Test chat application with Mockito

Given a simple chat application with the following classes:

```
public class MessageSender
{
     public String send(final String string)
     {
          return "- O K -";
     }
}

public class ChatEngine
{
    private final MessageSender messageSender;

    public ChatEngine(final MessageSender messageSender)
    {
        this.messageSender = messageSender;
    }

    public String say(final String message)
    {
        return messageSender.send(message);
    }
}
```

The `ChatEngine` class is now to be extended by a test that responds with the text "`SERVICE`" when `say("SECRET")` is called.

Exercise 5: Abstract database access with Mockito

Given the following class `Ex05_PersonService`. This uses the `PersonDAO` class to access a database and returns the `Person` domain class.

```
public class Ex05_PersonService
{
    private final PersonDao dao;

    public PersonService(PersonDao dao)
    {
        this.dao = dao;
    }

    public List<Person> findAll()
    {
        return dao.findAll();
    }

    public Person findById(int id)
    {
        return dao.findById(id);
    }
}


public class PersonDao
{
    public List<Person> findAll()
    {
        return Collections.emptyList();
    }

    public Person findById(int id)
    {
        return null;
    }
}
```

In order to be able to express unit tests without dependency or running connection to the database, the `PersonDAO` has to be simulated suitably with Mockito.

Exercise RETURN-VALUE: Write a test that returns a valid person for a invocation of the `findById(1)` method. What happens if you query for id 2?

Exercise EXCEPTION: Write a test that should check the error case of an invalid Id. For negative Ids an `IllegalArgumentException` should be thrown.

Exercise METHOD-CALLED: Write a test that checks whether a special method has been called. The desired result should be a list with two persons.

Exercise CALL-COUNT: Write a check for the number of calls for the following code fragment for `findById()` and `findAll()` — check for exact match for Id 1 only,

```
// Act
final Person person11 = service.findById(11);
if (person11 == null)
{
    final List<Person> result = service.findAll();
    final Person person1 = service.findById(1);
    final Person person2 = service.findById(2);
}
```

Tip: Use `anyInt()` as placeholder for `findById()`.

Exercise 6: Mockito - Date Conversion & Mock Injection

Given the following rudimentary unfinished class. `Arabic2RomanConverter`

```
public class Arabic2RomanConverter
{
     public String convert(int i)
     {
          return "" + i;
     }
}
```

This is used in a conversion service that converts a `LocalDate` into a representation with Roman numerals, i.e. from 7.2.1971 the text VII-II-MCMLXXI is generated.

```
public class DateInRomanCharsService
{
   private Arabic2RomanConverter converter;

   public String getRomanDate(final LocalDate date)
   {
        String day = converter.convert(date.getDayOfMonth());
        String month = converter.convert(date.getMonthValue());
        String year = converter.convert(date.getYear());

       return String.format("%s-%s-%s", day, month, year);
   }
}
```

Write a test `DateInRomanCharsServiceTest` that both mocks the converter appropriately and generates the appropriate mocks using annotations.

PART VII/VIII: Test Smells / Test Coverage — 45 — 60 min

Exercise 1: Test Smells & Test Elegance

Examine the class `Ex01_VersionNumberUtilsTest` for possible test smells. Also simplify too complex tests and use features introduced with JUnit 5. Try for example AssertJ functionality similar to

```
assertThat(mike).usingComparator(byAge).isGreaterThan(tim);
```

to get a better feel for JUnit 5 and for clean testing.

Use the following method signature as a starting point:

```
private void assertComparator(IntPredicate expectedResult,
                              int compareResult,
                              String v1, String v2, String resultHint)
{
    // TODO
}
```

This is intended to produce error messages that speak for the intentionally erroneous inputs, such as:

```
or: Multiple Failures (2 failures)
comparing 3.5 with 3.1.72 should result in 3.5 < 3.1.72 ==> expected: <true> but was: <false>
comparing 3.1.72 with 3.5 should result in 3.1.72 > 3.5 ==> expected: <true> but was: <false>
```

Tip: Use the following Predicates:

```
IntPredicate IS_SAME = n -> n == 0;
IntPredicate IS_SMALLER_THAN = n -> n < 0;
IntPredicate IS_BIGGER_THAN = n -> n > 0;
```

Bonus: Use AssertJ and change tao use `Comparator<E>`.

Exercise 2: Test Order

After performing a short analysis, the given test class `Ex02_StudyGroupTest` obviously tests a sequence of actions. Split these into suitable smaller test methods so that each of these tests one aspect.

Tip: Use `@TestMethodOrder`.

Exercise 3: Test Coverage

Given the following class `Ex03_ScoreCalculator`. Write test cases for it so that a test coverage of at least 70 and finally 100 % is achieved.

```
public class Ex03_ScoreCalculator
{
      public static String calcScore(int score)
      {
            if (score < 45)
                  return "failed";
            else
            {
                  if (score > 95)
                        return "passed with distinction";

                  return "passed";
            }
      }
}
```

Exercise 4: Test Coverage

Given the following class `Ex04_ArrayUtils` with a faulty implementation. Write test cases for it so that a test coverage of at least 80 and 100 % is achieved.

```
public class Ex04_ArrayUtils
{
    public static int indexOf(int[] values, int searchFor)
    {
        if (values == null)
           throw new IllegalArgumentException("Array is null");

        int index = -1;
        for (int i = 0; i <= values.length; i++)
        {
            if (values[i] == searchFor)
            {
                index = i;
                break;
            }
        }
        return index;
    }
}
```

Exercise 5: Test Coverage

Given the following class `Ex05_GreetingCreator` with a faulty implementation. Write test cases for it, so that a test coverage of at least 80 and finally 100% is reached.

```
public class Ex05_GreetingCreator
{
    public String createGreeting(final LocalTime time)
    {
        String message = "Good ";
        if (time.isBefore(LocalTime.of(12, 0, 0)))
        {
            message += "Morning";
        }
        else if (time.isAfter(LocalTime.of(12, 0, 0)))
        {
            message += "Afternoon";
        }
        else if (time.isAfter(LocalTime.of(18, 0, 0)))
        {
            message += "Evening";
        }

        return message;
    }
}
```