Software Testing Advanced Features

Agenda

1. JUnit5
2. Mockito

JUnit5

JUnit5

JUnit5 is **a framework** for **testing Java code**. It is the next generation of JUnit. This includes focusing on **Java 8 and above**, as well as enabling **many different styles of testing**. It is (as well as JUnit itself) an **open source** project which can be easily **extended** by **other frameworks**
such as:

- Hamcrest
- Mockito
- PowerMock

Maven Snippet

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.6.2</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.6.2</version>
    <scope>test</scope>
</dependency>
```

JUnit5 – Repeating test

Example

```
@RepeatedTest(3)
void shouldReturnAddRepeat(){
    //given
    Calculator calc = new Calculator();

    //when
    double result = calc.add(15, 8);

    //then
    assertEquals(23, result);
}
```

Exercises – JUnit5

1. Write tests for the exercises from JDBC and Hibernate block. Use repeating test for CRUD operations. Check how to properly handle exceptions. E.g. you shouldn't be able to:
    - Delete non existing value (first delete – OK, second – NO)
    - Insert the same row, with some unique field, twice
    - Update some row with the same values (optionally it may pass)

JUnit5 – assertions

Values can be **asserted** with the usage of multiple **static methods**.

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

JUnit5 – assertions

Example

```
assertEquals(64, 2 * 32);
assertEquals(1, 2, "Values are not equal");
assertTrue(condition);
assertFalse(condition);
assertArrayEquals(array1, array2);
assertIterableEquals(list1, list2);
assertNull(object);
assertNotNull(object);
assertSame(object1, object2);
```

JUnit5 – grouping assertions

Example

```
@Test
void shouldReturnMultiplicationOperation(){
    assertAll(
        () -> assertEquals(4, calculator.multiply(2,2)),
        () -> assertEquals(81, calculator.multiply(9,9)),
        () -> assertEquals(30, calculator.multiply(5,6))
    );
}
```

JUnit5 – Exceptions

Exceptions can be tested (and then examined) using the **assertThrows()** method.

JUnit5 – Exceptions

Example

```
@Test
void shouldAcceptDivideByZero(){
    IllegalArgumentException exception =
        Assertions.assertThrows(IllegalArgumentException.class,
        () -> calculator.divide(10,0));

    assertEquals("Divide by 0", exception.getMessage());
}
```

JUnit5 – Lifecycle

If we want some instructions to be **run before or after each method**, we can put a method in the test class with ***@BeforeEach*** or ***@AfterEach*** **annotation**.

On the other hand, if we would like some methods to be run **only once** before launching all of the tests cases within the Test Class. We can use ***@BeforeAll*** or ***@AfterAll*** **annotation**.

JUnit5 – BeforeEach and AfterEach

Example

```
class TestClass {

    @BeforeEach
    void setUp() {
        System.out.println("Run before each test");
    }

    @AfterEach
    void tearDown() {
        System.out.println("Run after each test");
    }

    // @Test annotated methods
    ...
}
```

JUnit5 – BeforeAll and AfterAll

Example

```
class TestClass {

    @BeforeAll
    static void setUpTestCase() {
        System.out.println("Run before the first test method")
    }

    @AfterAll
    static void tearDownTestCase() {
        System.out.println("Run after the last test method");
    }
    // @Test annotated methods
    ...
}
```

JUnit5 – Parameterized tests

Parameterized tests are like other tests except that we add the @ParameterizedTest annotation and one of the following:

- @ValueSource
- @MethodSource
- @ArgumentsSource
- @CsvSource
- @CsvFileSource
- @EnumSource

JUnit5 – Parameterized tests

Example

```
@ParameterizedTest
@ValueSource(doubles = {10.0, -23.0, 12.0, -2.0})
void shouldReturnReversedSign(double a){
    assertEquals(-1 * a, calculator.reverseSign(a));
}
```

```
@ParameterizedTest
@MethodSource(names = "getParameters")
void shouldReturnReverseSign(double a){
    assertEquals(-1 * a, calculator.reverseSign(a));
}

static Stream<Arguments> getParameters(){
    return Stream.of(Arguments.of(1.0),
        Arguments.of(-231.0),
        Arguments.of(26.0),
        Arguments.of(-98.0));
}
```

JUnit5 – Parameterized tests

Example

```
@ParameterizedTest
@ArgumentsSource(Parameters.class)
void shouldReturnReverseSing3(double a){
    assertEquals(-1 * a, calculator.reverseSign(a));
}

static class Parameters implements ArgumentsProvider {
    @Override
    public Stream<? extends Arguments> provideArguments(final ExtensionContext context) {
        return Stream.of(Arguments.of(23.0),
            Arguments.of(-56.0),
            Arguments.of(64.92),
            Arguments.of(0.32));
    }
}
```

JUnit5 – Parameterized tests

Example

```
@ParameterizedTest
@CsvSource({"10.0, 10.0, 20.0", "13.4, 2.9, 16.3", "123.2, 5.3, 128.5"})
void shouldReturnAdditionOperation(double a, double b, double sum){
    assertEquals(sum, calculator.add(a, b));
}
```

```
@ParameterizedTest
@CsvFileSource(resources = "/dataSource.csv")
void shouldReturnSubtractionOperation(double a, double b, double sub){
    assertEquals(sub, calculator.sub(a, b));
}
```

JUnit5 – Argument Conversion

JUnit5 converts the **String** arguments to **the specified enum type**. To support use cases like this, JUnit Jupiter provides **a number of built-in implicit type converters**.

```
@ParameterizedTest
@ValueSource(strings = {"APRIL", "JUNE", "SEPTEMBER", "NOVEMBER"})
void someMonths_Are30DaysLong(Month month) {
    final boolean isALeapYear = false;
    assertEquals(30, month.length(isALeapYear));
}
```

JUnit5 – EnumSource

JUnit5 allows to provide enum values as arguments with @EnumSource. It is possible to include or exclude some values (or use all of them)

```
@ParameterizedTest
@EnumSource(value = Month.class, names = {"APRIL", "JUNE", "SEPTEMBER",
"NOVEMBER"}, mode = EnumSource.Mode.INCLUDE)
void someMonthEnums_Are30DaysLong(Month month) {
    final boolean isALeapYear = false;
    assertEquals(30, month.length(isALeapYear));
}
```

JUnit5 – Explicit Conversion

In addition, JUnit5 also gives us the opportunity to provide a **custom and explicit converter for arguments**.

JUnit5 – Explicit Conversion

Example

```
@ParameterizedTest
@ValueSource(strings = {"A", "D", "16"})
void shouldReturnIntToHex(@ConvertWith(HexToInt.class) int a){
    assertEquals(-1 * a, calculator.reverseSign(a));
}

static class HexToInt extends SimpleArgumentConverter {
    @Override
    protected Object convert(Object o, Class<?> targetType){
        assertEquals(int.class, targetType, "Can only convert to int");
        return Integer.decode("0x" + o.toString());
    }
}
```

Exercises – JUnit

1. Create simple Employee class. Write common setters and getter for fields like salary, age and so on.
    - Using parametrized tests write boundary tests (like negative salary, minimal positive salary, negative age).
    - Check if you can set e-mail address in wrong format.
    - Check if you can set phone number in wrong format.

Mockito

Mockito

A mock is **a substitution** for **a real object**.
It can be used to mock (fake) it's methods or to verify methods executions (among others).

For the Mockito mocks to work the test class has to be annotated:

```
@ExtendWith(MockitoExtension.class)
public class MockExampleTest {
    // tests ...
}
```

Maven Snippet

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.6.2</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>3.4.6</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>3.4.6</version>
    <scope>test</scope>
</dependency>
```

Example

```
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
class CarRentalTest {
    @Mock
    private Agency agencyMock;

    @Test
    void verifyExample1() {
        agencyMock.findCar(5, "sedan");
        verify(agencyMock).findCar(5, "sedan");
    }
}
```

Example

```
@Test
void verifyExample2() {
  agencyMock.findCar(5, "sedan");
  verify(agencyMock, atMost(10)).findCar(any(Integer.class), anyString());
}
```

```
@Test
void verifyExample3() {
    agencyMock.findCar(5, "sedan");
    verify(agencyMock, atLeastOnce()).findCar(eq(5), startsWith("sed"));
}
```

Example

```
@Test
void whenExample1() {
    List<Car> cars = agencyMock.findCar(5, "sedan");
    assertNotNull(cars);
    assertTrue(cars.isEmpty());
}
```

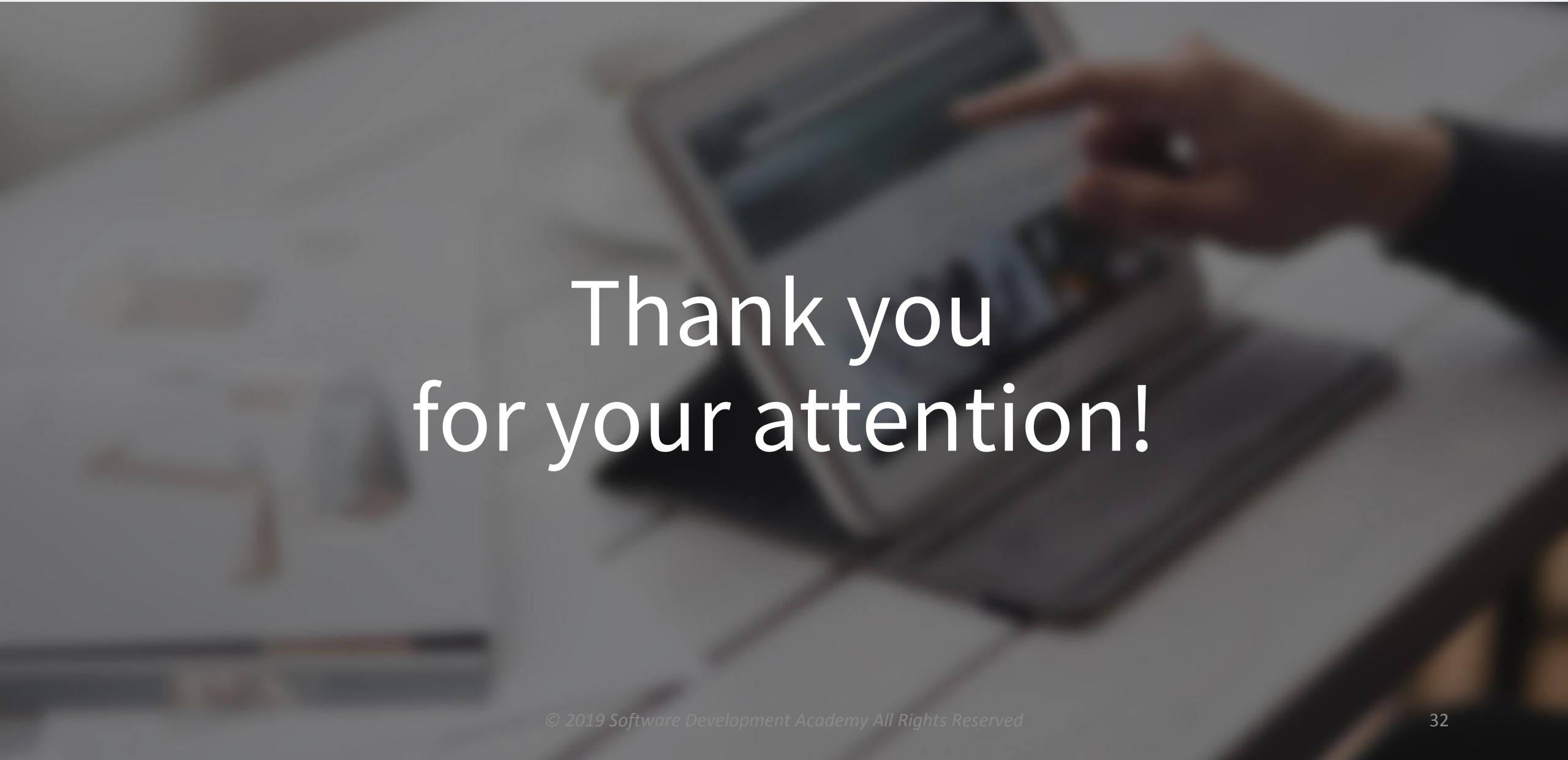
```
@Test
void whenExample2() {
    Car someCar = new Car("Ford", "Focus", 2.0, 5, "sedan", 100.0);
    when(agencyMock.findCar(5, "sedan"))
        .thenReturn(Collections.singletonList(someCar));

    List<Car> flight = agencyMock.findCar(5, "sedan");
    assertEquals(1, flight.size());
    assertEquals(someCar, flight.get(0));
}
```

Exercises

Choose one of the following exercises:

1. Write a Mock for a database interface. Support common CRUD operations.
2. *Write a Mock for an embedded device. Consider that you have to communicate with it using serial port. Use an interface to simplify the mocking process. Mock should send a heartbeat every couple of seconds and answer for some basic commands, like „blink led”, „get state”. Mock may receive and return data in JSON format.



Thank you for your attention!