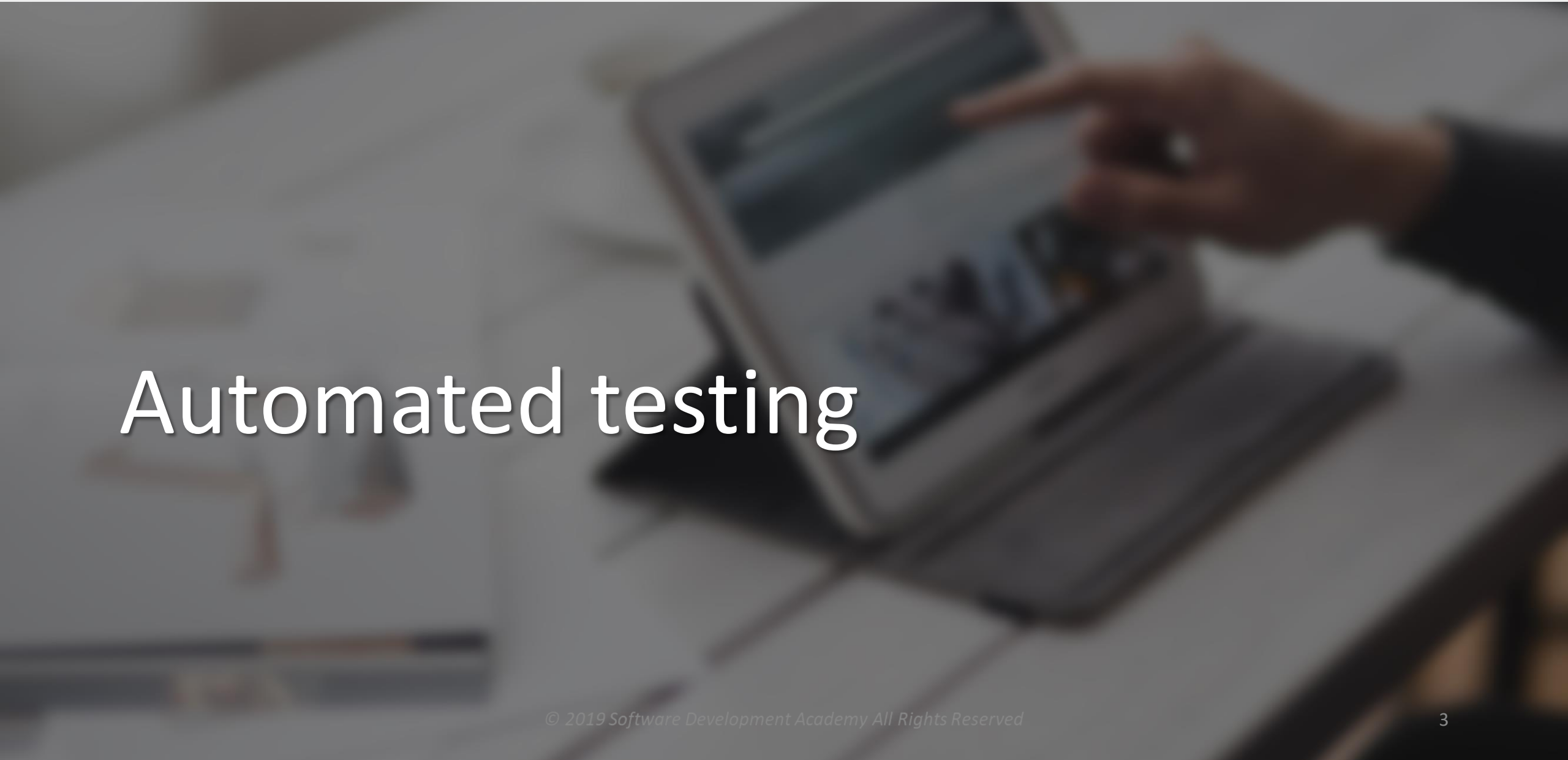
Software Testing

Agenda

1. Automated testing
    a) JUnit
    b) Mockito
    c) AssertJ
2. TDD – Test Driven Development



Automated testing

Automated testing

- **Decreases time** required to test code.
- **Reduces** required human **resources**.
- Is more **reliable**.
- Can be **programmed**.

JUnit

JUnit

JUnit is **a framework** for **testing Java code**. Although fifth version has been released, most commonly used one is the **fourth generation**. It is an **open source** project which can be easily **extended** by **other frameworks** such as:

- Hamcrest
- Mockito
- PowerMock

Example

```
import org.junit.Test;

public class ExampleTest {
    @Test
    public void test() {
        // instructions
    }
}
```

Junit – given/when/then

Test should be divided into **three sections**:

- **given** - preset objects for the test,
- **when** - an action to be tested is performed,
- **then** - testing if action performed correctly.

Junit – test results

Test has **passed** if **no** ***Exception*** is **thrown**.
If an ***AssertionError*** is thrown it will result with **failure**.
Exception of any other type results with an **error**.

Junit – given/when/then and assert keyword

Example

```
@Test
public void test() {
    // given
    Calculator calculator = new Calculator();
    // when
    int result = calculator.add(5, 3);
    // then
    assert result == 8;
}
```

Maven Snippet

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.XXX</version>
    <scope>test</scope>
</dependency>
```

Exercises – JUnit

1. Create Calculator Class. It should have two methods:
    a) add(int x, int y)
    b) sub(int x, int y)
2. Write at least one test for every method from Calculator Class (use *assert* keyword).

Junit – assertions

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

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

Junit – assertions

Example

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

Junit – Exceptions

If **an exception** is **expected** for a test to be run correctly we can add **expected parameter** to the **@Test annotation**.

Junit – Exceptions

Example

```
@Test(expected = ArithmeticException.class)
public void shouldThrowExceptionWhenDividingBy0() {
    // given
    Calculator calculator = new Calculator();
    int number = new Random().nextInt();
    // when
    calculator.divide(number, 0);
    // then
    // should throw exception
}
```

Junit – Exceptions Rule

Exceptions can also be caught by adding **a public field of type ExpectedException** annotated with ***@Rule***.

Junit – Exceptions Rule

Example

```
public class CalculatorTest {
    @Rule
    public ExpectedException expectedException = ExpectedException.none();

    @Test
    public void shouldThrowExceptionWhenDividingBy0() {
        // give
        expectedException.expect(ArithmeticException.class);
        expectedException.expectMessage("/ by zero");
        Calculator calculator = new Calculator();
        int number = new Random().nextInt();
        // when
        calculator.divide(number, 0);
        // then
        // should throw expected exception
    }
}
```

Junit – Lifecycle

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

Junit – Before and After

Example

```
public class TestClass {

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

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

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

Junit – BeforeClass and AfterClass

Example

```
public class TestClass {

    @BeforeClass
    public void setUpBeforeClass() {
        System.out.println("Run before the first test method")
    }

    @AfterClass
    public void tearDownAfterClass() {
        System.out.println("Run after the last test method");
    }
    // @Test annotated methods
    ...
}
```

Junit – Parameterized tests

If you want to run the same test for multiple parameters combinations, you can simply annotate the test class:

```
@RunWith(Parameterized.class)
public class ParametrizedTest {
    ...
}
```

Junit – Parameterized tests

Example

```
@Parameters
public static Collection<Integer[]> parameters() {
    return Arrays.asList(new Integer[][] {
        {1, 1, 2},
        {2, 4, 6},
        {1, 6, 7},
        {4, 1, 5}
    });
}
```

```
@Parameter(0)
public int argument1;

@Parameter(1)
public int argument2;

@Parameter(2)
public int expectedResult;
```

Exercises – JUnit

1. Think about more functional tests for Calculator Class – what will happen, if both Integer values will contain maximum values (Integer.MAX_VALUE)?
2. Add divide and multiply methods, think about tests, that should be written for that case.

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:

```
@RunWith(MockitoJUnitRunner.class)
public class MockExampleTest {
    // tests ...
}
```

Maven Snippet

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.10.19</version>
    <scope>test</scope>
</dependency>
```

Example

```
import static org.mockito.Mockito.mock;

Calculator calculator = mock(Calculator.class);
```

```
@Mock
private Calculator calculator;
```

```
when(calculator.add(eq(1), eq(3))).thenReturn(123);
// from now on calculator.add(1, 3) returns 123.
```

```
when(calculator.add(anyInt(), anyInt())).thenReturn(123);
```

```
verify(calculator).add(anyInt(), anyInt());
```

Mockito – spying on an object

Example

```
Calculator calculator = new Calculator();
Calculator spyCalculator = spy(calculator);

when(spyCalculator.add(eq(4), eq(6))).thenReturn(55);

Calculator calculator = new Calculator();
Calculator spyCalculator = spy(calculator);
spyCalculator.add(2, 5);
verify(spyCalculator).add(anyInt(), anyInt());
```

AssertJ

AssertJ

AssertJ is a tool that **allows** the usage of **fluent assertions** in java.
By fluent we mean **chained method invocation**.

To be sure of using AssertJ methods such import line can be put into a class:

```
import static org.assertj.core.api.Assertions.*;
```

Maven Snippet

```
<dependency>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-core</artifactId>
    <version>3.9.1</version>
    <scope>test</scope>
</dependency>
```

Example

```
String text = "abc";
assertThat(text).isEqualTo("abc");

String[] stringArray = {"abc", "cde", "efg"};
assertThat(stringArray)
        .hasSize(3)
        .contains("cde")
        .doesNotContain("xyz");
```

TDD – Test Driven Development

Test Driven Development

Test Driven Development is a process that **starts** with **writing tests** and then **implementing methods** for the **tests to pass**.

The test **should fail** as long as tested method won't be implemented properly.

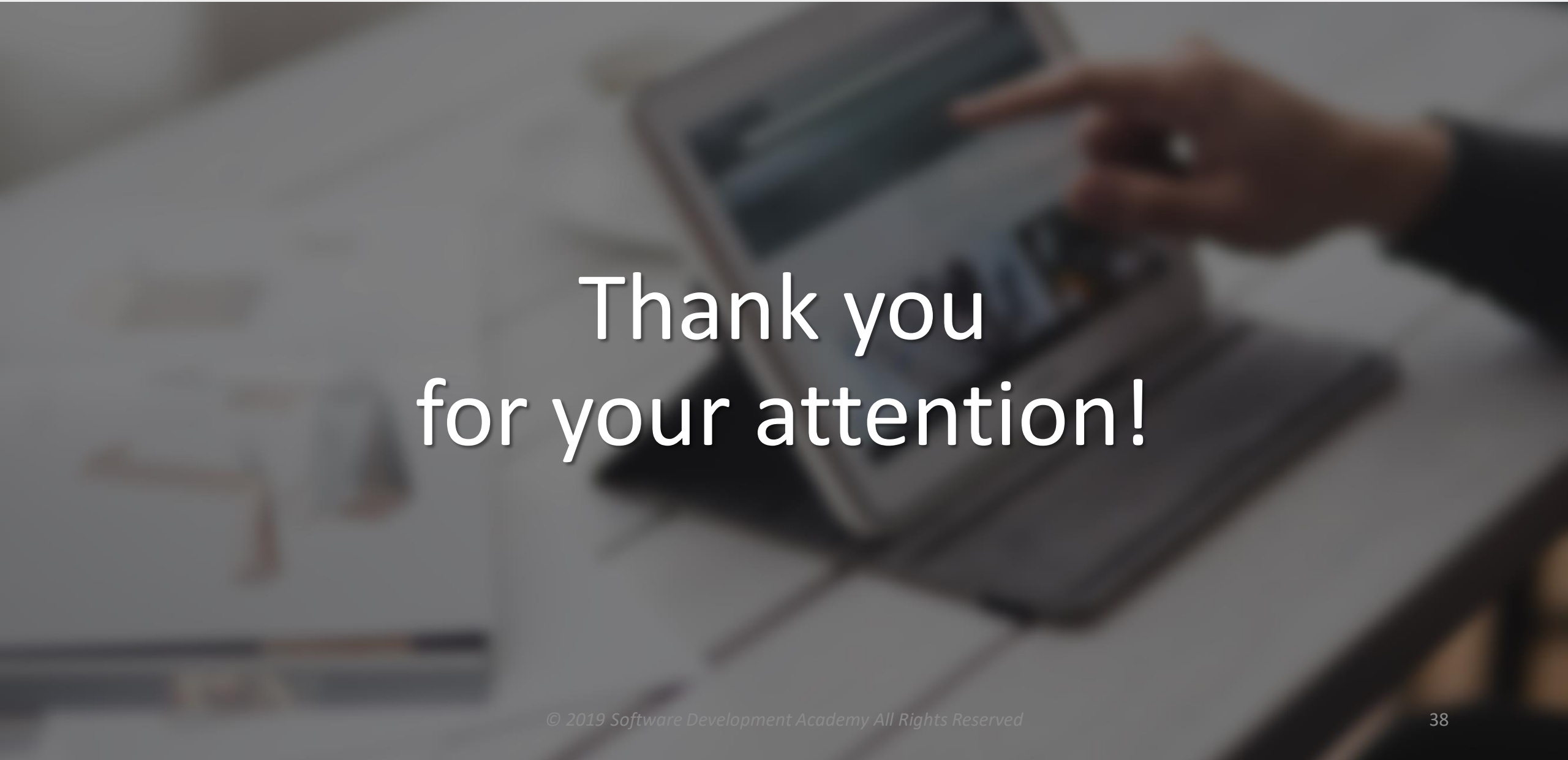
Test Driven Development

Test Driven Development can be divided into three phases:

- Red
- Green
- Refactor

Exercises

1. Create Person Class. It should have one method:
    a) boolean isTeenager() -> returns True if Person is older than 10 and younger than 20
2. Write tests to verify every boundary value (e.g. 10, 11, ..).
3. Create FahrenheitCelciusConverter Class. It should contain two methods:
    a) toCelcius(int fahrenheit)
    b) toFahrenheit(int celcius)
4. Write at least two tests for every method.
5. Mock FahrenheitCelciusConverter Class. Mocked-up methods should print their names and exit.



Thank you for your attention!