Software Testing

Agenda

1. What and why do we test?
2. Testing with JUnit
3. Using fluent assertion with AssertJ
4. Test Driven Development basics

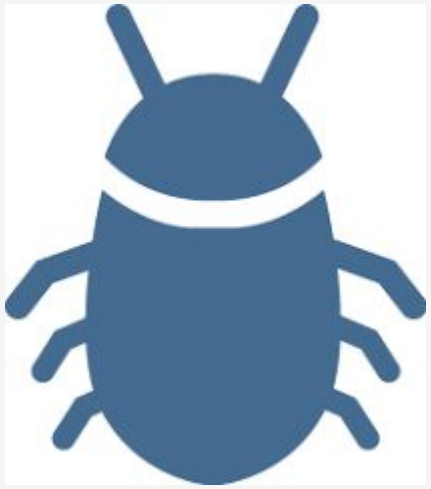
What is testing?

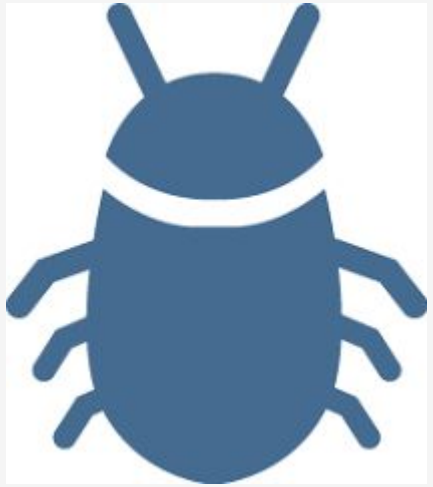
- **testing** is **one** of the process of creating any application

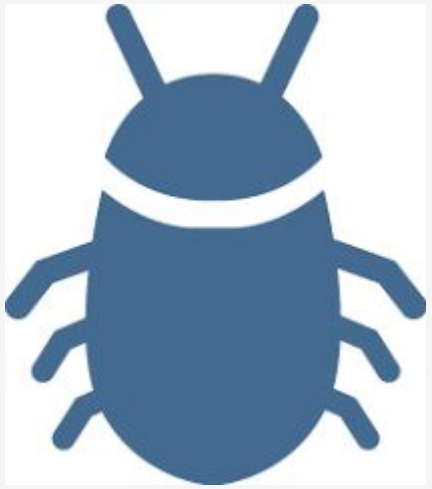
- process that verifies if software we create works as expected
  - is the result just what we expect?
  - does the functionality match the specification and requirements?

Bugs

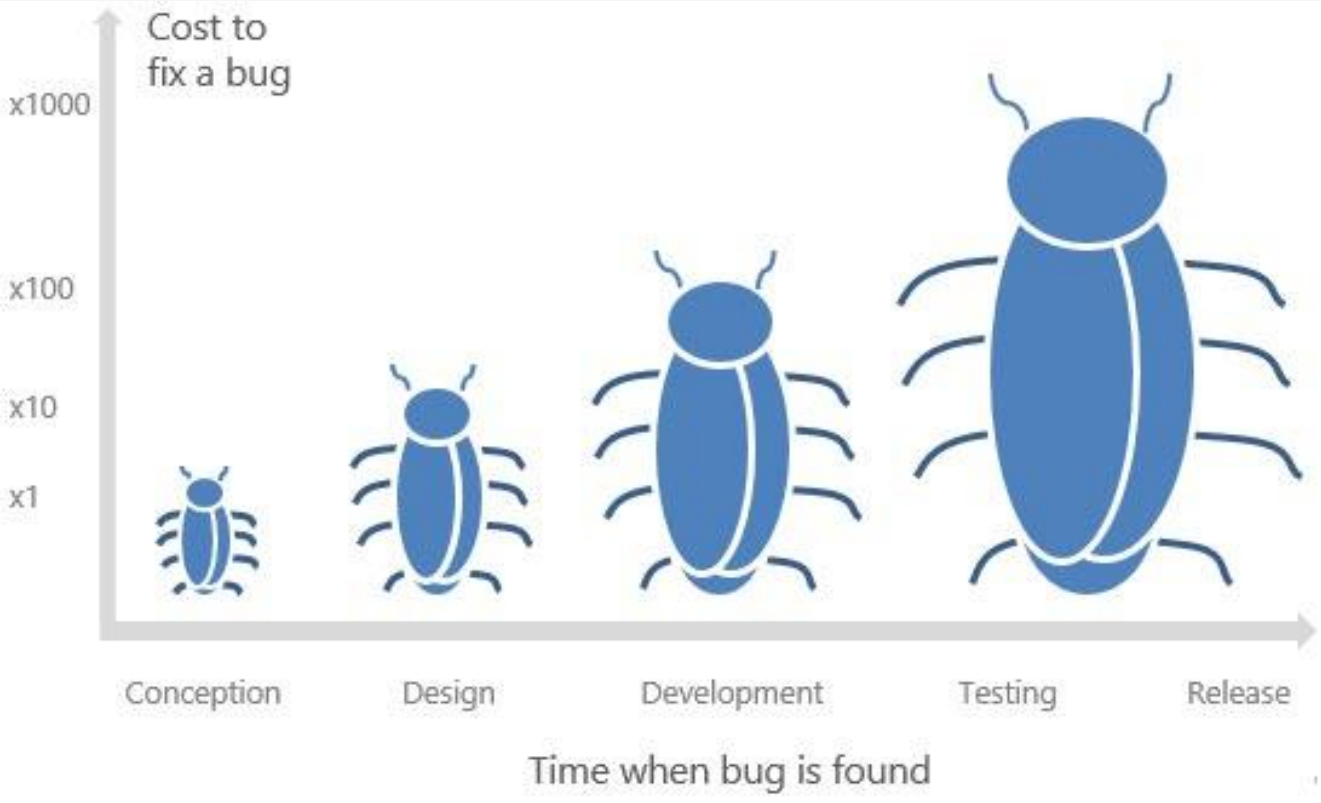
- **Bugs** are **one of** the potential result of running tests
  - software errors







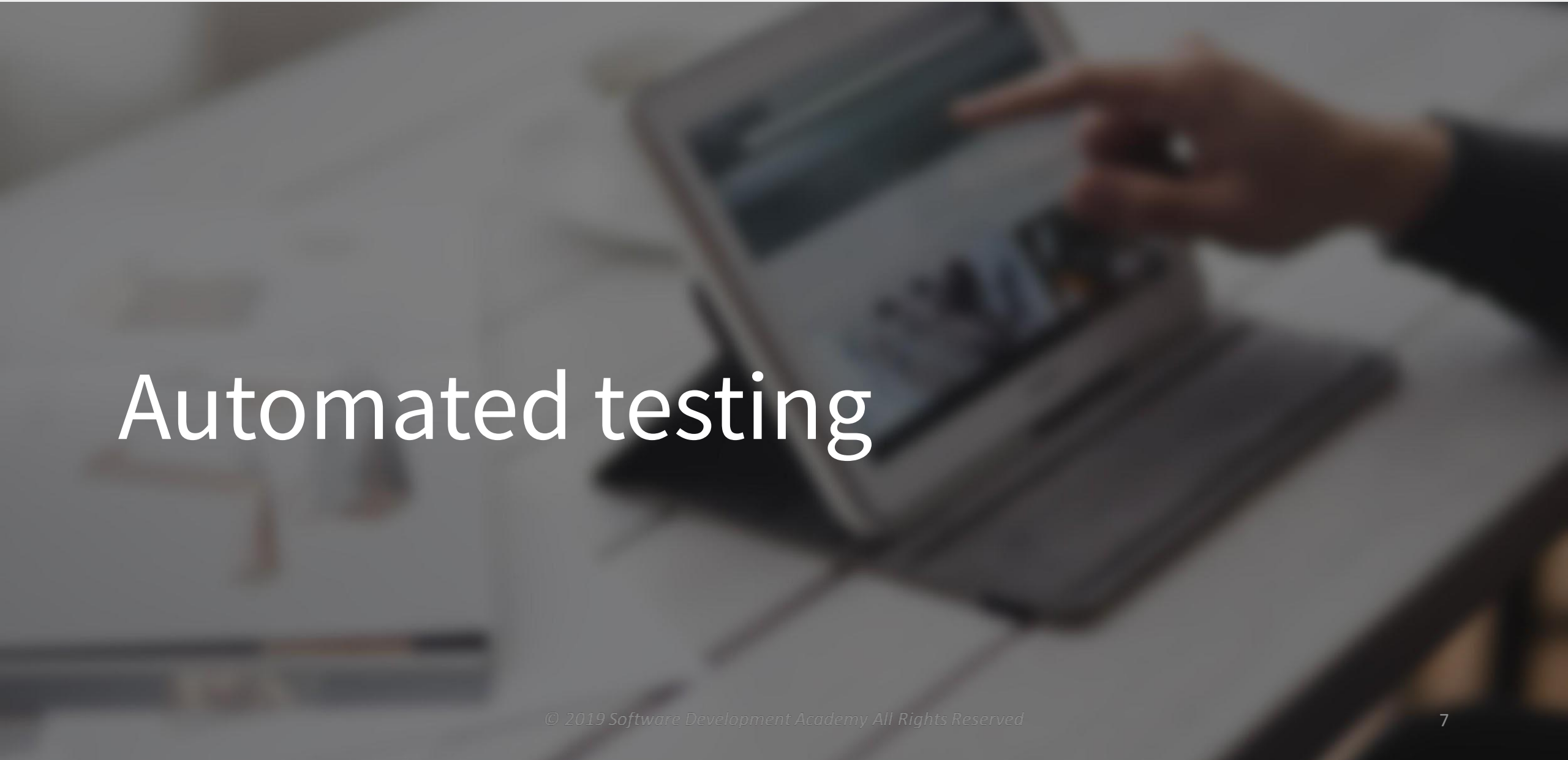
Bug cost



Test types

- manual tests
  - each time scenario needs to be manually executed
  - each time tester needs to verify if the result is as expected

- automated tests
  - written once
  - can be easily repeated
  - expected result checked automatically



Automated testing

Test types

- unit tests
- integration tests
- functional tests (end to end tests)

- other (performance, stress, contract etc.)

Unit tests

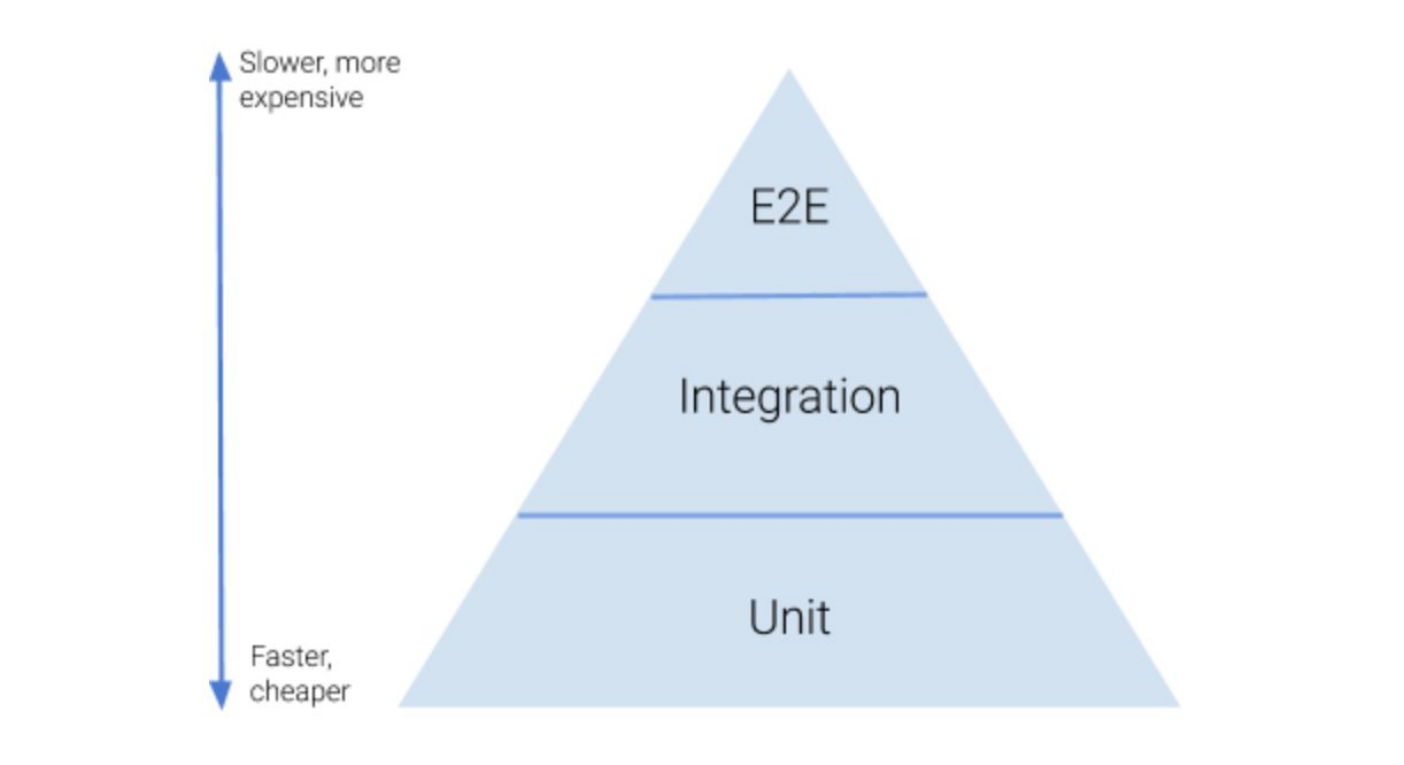
- test checks isolated element (method in a class)
- one possible failure
- quick execution (< 1s)
- no configuration needed

Integration tests

- test executed with some external dependencies (e.g. database)
- many potential reasons to fail
- usually longer execution than unit tests
- some configuration sometimes needed depending on environment

UNIT TESTS AND INTEGRATION TESTS ARE OFTEN CREATED BY DEVELOPERS

Tests - type vs time and quantity



Good unit tests - FIRST

- **Fast** - low execution time
- **Isolated/independent** - one test should not depend on state that was prepared by other test
- **Repeatable -** tests should always give same results
  - avoid date, time, random numbers, checking order of HashSet etc.
- **Self-validating** - tests should check the actual vs expected result
- **Through/timely -** tests should check most/all possible scenarios
  - consider positive scenarios, negative scenarios (e.g. empty String) and exceptions thrown in the code

Unit testing - libraries

- There are multiple testing libraries

- we will focus on **JUnit 5**

JUnit

JUnit

- Library for creating tests in Java
- One of the most popular libraries in Java
- Used versions: 4 and 5
- We focus on JUnit**5**

In order to use JUnit in project add following dependency:

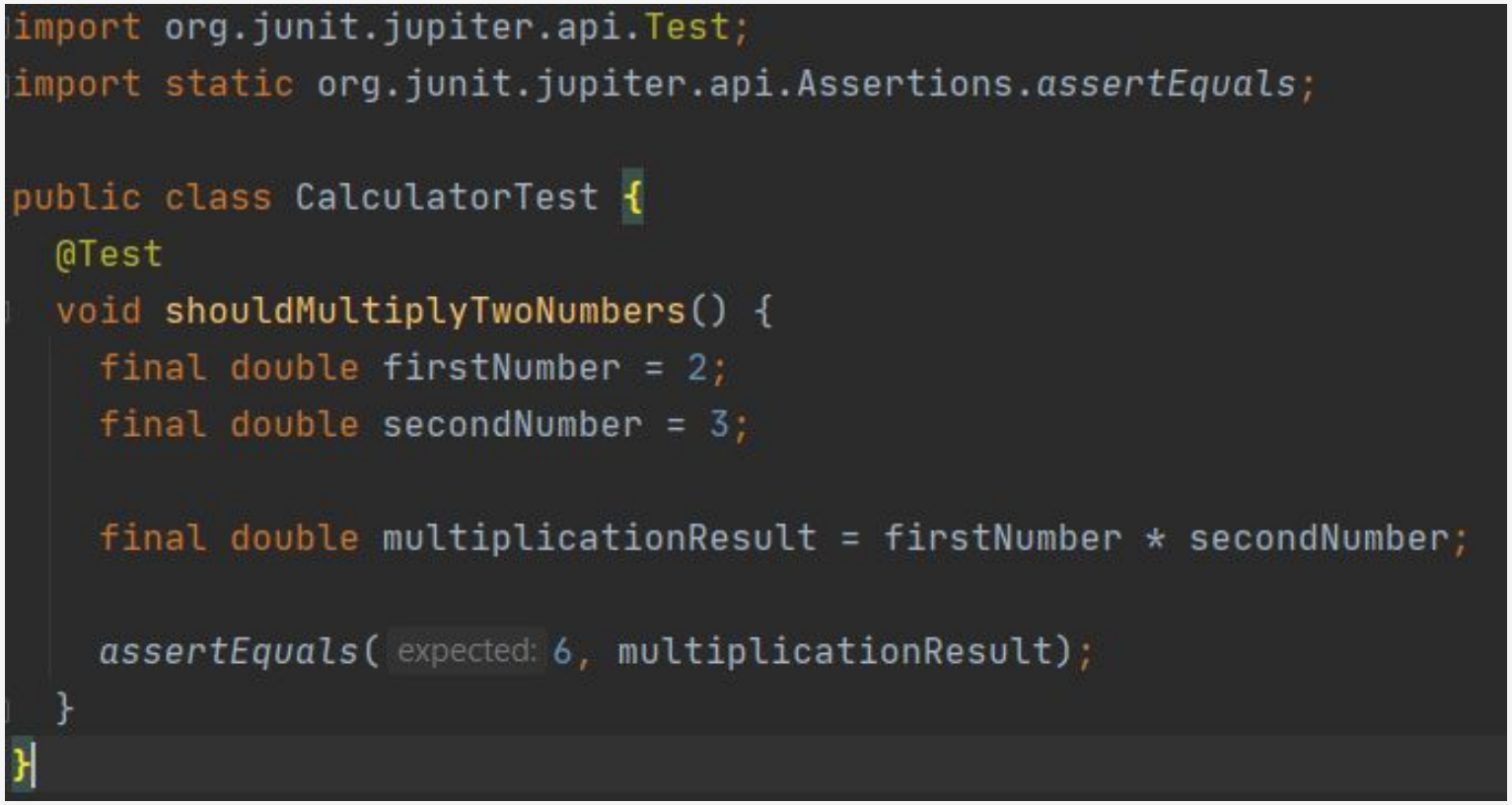
```
<dependency>
   <groupId>org.junit.jupiter</groupId>
   <artifactId>junit-jupiter-engine</artifactId>
   <version>5.7.0</version>
   <scope>test</scope>
</dependency>
```

newer version may be available

JUnit tests

- Class name - usually for class X, XTest contains tests for class X
- test is any method in class annotated with @Test annotation
  - no public modifier needed
- test is split into **3** parts
  - given
    - prepare all objects needed for test
  - when
    - execute method you want to test, assign the return value to a variable in case method returns a value
  - then
    - check expected vs actual values using assertions

JUnit - creating tests



```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class CalculatorTest {
  @Test
  void shouldMultiplyTwoNumbers() {
    final double firstNumber = 2;
    final double secondNumber = 3;

    final double multiplicationResult = firstNumber * secondNumber;

    assertEquals( expected: 6, multiplicationResult);
  }
}
```

JUnit - assertions

- assertEquals()
- assertTrue()/assertFalse()
- assertNotNull()/assertNull()
- assertSame()/assertNotSame()
- fail()
- assertArrayEquals()
- assertIterableEquals()
- assertLinesMatch()
- assertAll()

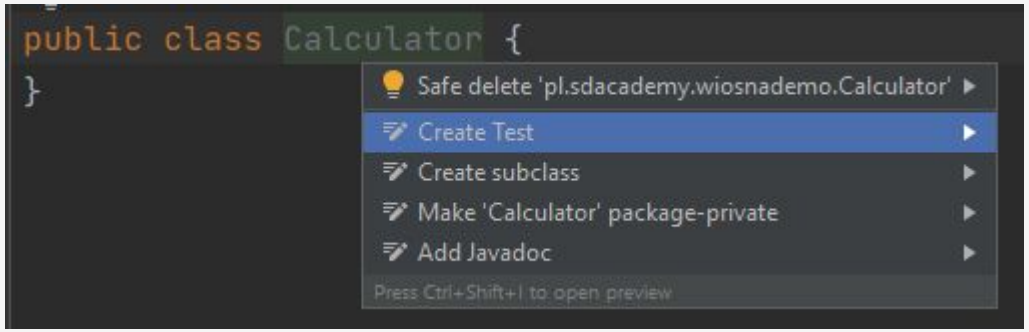
- and more...

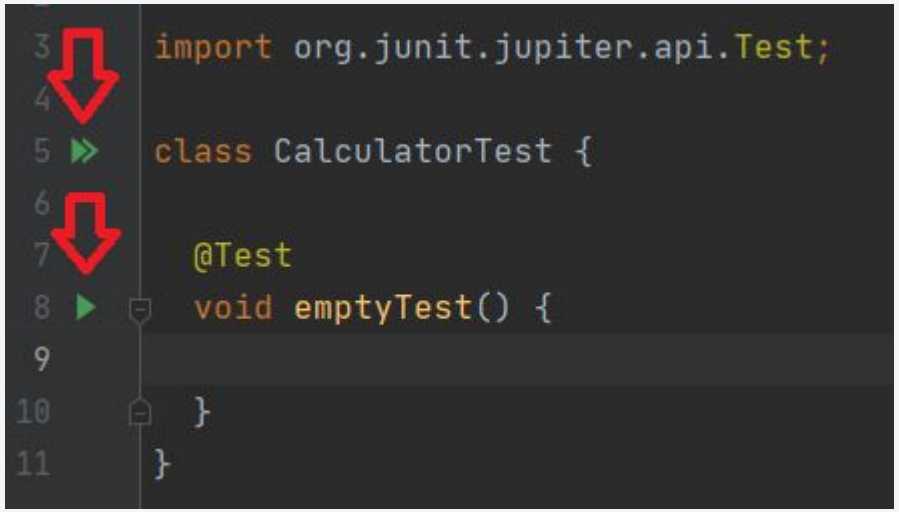
JUnit5 – assertions

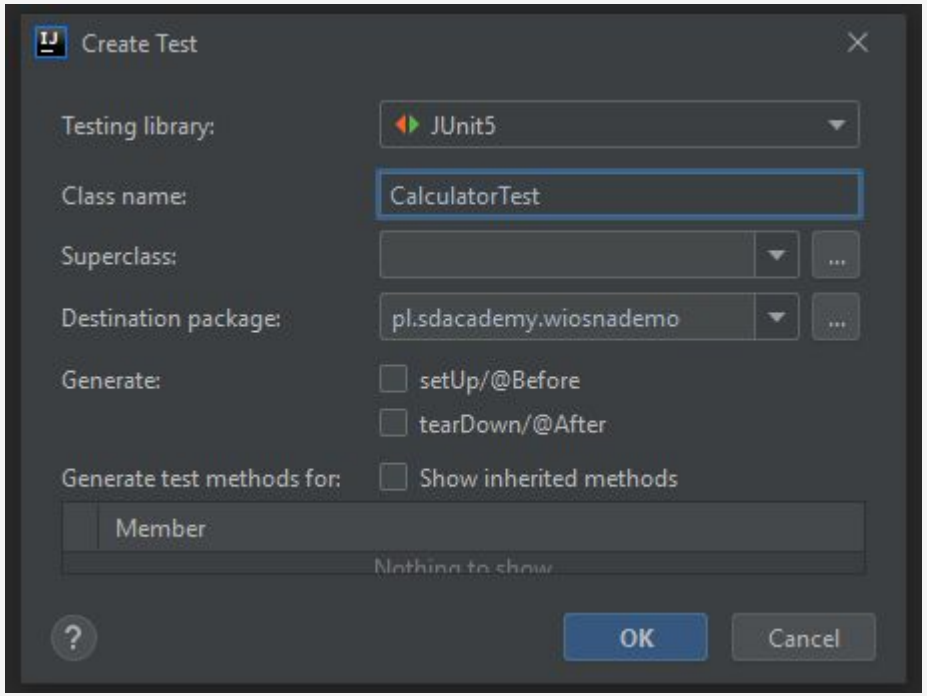
Example

```
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);
```

JUnit - creating and running tests







JUnit - lifecycle methods

- What if you need to create an instance of a class in every test in your class?
- What if you need to set some things up before all tests are executed?

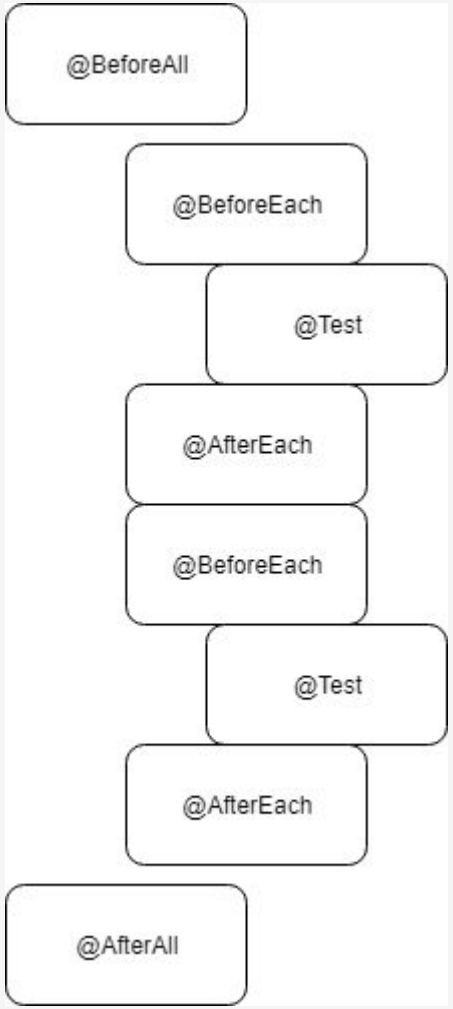
- JUnit offers **lifecycle methods** can be **optionally** defined (in any order) for each test class
  - @BeforeEach
  - @AfterEach
  - @BeforeAll
  - @AfterAll

JUnit - lifecycle methods

- defined on non static, **void** methods
  - @BeforeEach - run before every test
  - @AfterEach - run after every test

- defined on **static void** methods
  - @BeforeAll - run once before tests
  - @AfterAll - run once after tests

- all of those methods can have **any** name
- those methods do **not** have any arguments



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
    ...
}
```

JUnit - other functionalitites

- @DisplayName
  - can change test name
  - used when its easier to say what test is doing than providing description in method name
- @Disabled
  - turns off the test

JUnit - assertAll

- by default tests ends successfully or when **first** assertion fails
- *assertAll* - allows to group multiple assertions into **one** so all are always executed

- assertions need to be wrapped into “lambdas”

```
@Test
void shouldCreatePersonCorrectly() {
    final Person person = Person.create(FIRST_NAME, LAST_NAME);

    assertAll(
            () -> assertEquals(FIRST_NAME, person.getFirstName()),
            () -> assertEquals(LAST_NAME, person.getLastName())
    );
}
```

Advanced assertions - AssertJ

- Library that provides fluent assertions
- Gives access to many assertions
- Provides tools that increase readability (both tests and errors in case they occur)

- to use AssertJ include:

```
<dependency>
   <groupId>org.assertj</groupId>
   <artifactId>assertj-core</artifactId>
   <version>3.18.1</version>
   <scope>test</scope>
</dependency>
```

newer version may be available

Advanced assertions - AssertJ

AssertJ does **not** replace JUnit but uses different methods and syntax in **then** section
AssertJ allows to **chain** multiple assertions (call multiple methods after ‘dot’)

How to execute AssertJ assertions in **any** object?

assertThat(testedObject)

- static method
- different assertions available depending on type of tested object
- ```
  import static org.assertj.core.api.Assertions.assertThat;
  ```
  ```

AssertJ - examples

```java
@Test
void junitAssertions() {
    final String actual = "I_love_sda";

    assertEquals(EXPECTED, actual);
}
```

```java
@Test
void assertjAssertions() {
    final String actual = "I_love_sda";

    assertThat(actual).isEqualTo(EXPECTED);
}
```

```java
@Test
void assertJDemo1() {
  String actualResult = testedMethod();

  assertThat(actualResult) AbstractStringAssert<capture of ?>
      .startsWith("a") capture of ?
      .endsWith("b")
      .contains(" ");
}
```

```java
@Test
void assertJDemo2() {
  List<Integer> testedList = List.of(1, 2, 3, 4, 5);

  assertThat(testedList).hasSize(5)
      .containsAnyOf(5, 6, 7);
}
```

AssertJ - examples

Example assertions available for String:

- doesNotContainAnyWhitespaces
- isEqualTo
- containsPattern
- endsWith
- doesNotStartWith
- isBetween

AssertJ - examples

Example assertions available for List:

- isEqualTo
- contains
- containsAnyOf
- containsExactlyInAnyOrder
- isNotSameAs

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());
}
```

TDD – Test Driven Development

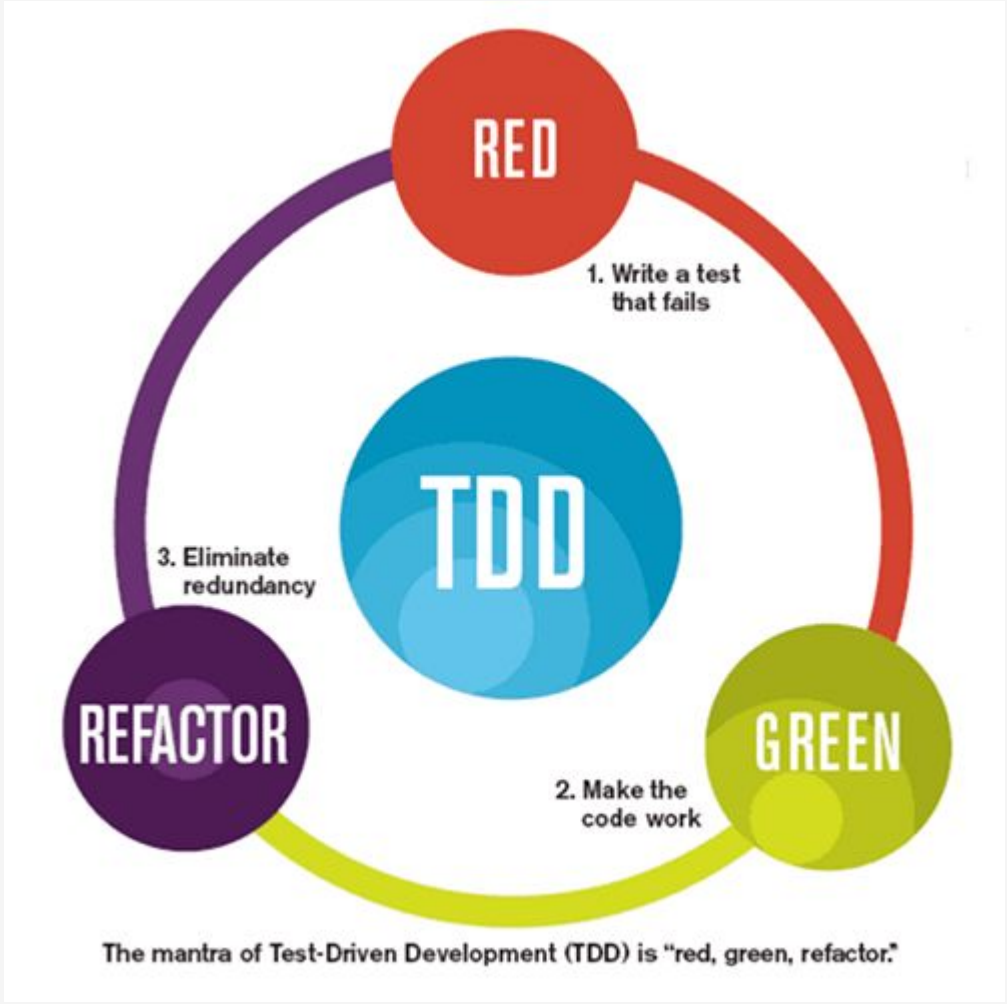
Test Driven Development

Test Driven Development can be divided into **three phases**:

- Red
- Green
- Refactor

TDD - Red Green Refactor

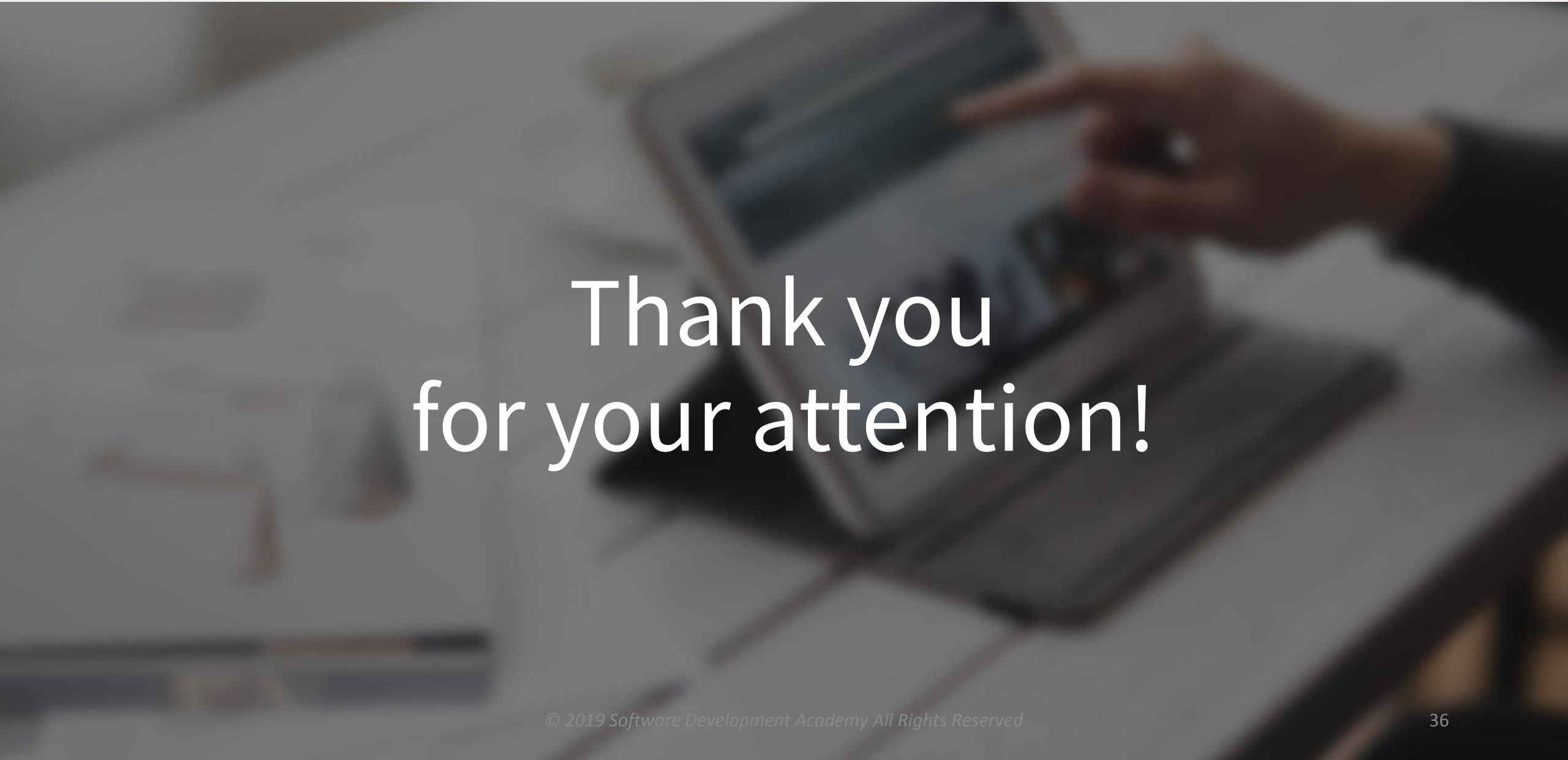
- red - write test that is failing
- green - create minimum implementation so that written tests are passing
- refactor - reorganize and make the code “clean”



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.



Thank you for your attention!