java from scratch Knowlage Base

Software Testing slides

Tests

Software testing is a process related to software development and software quality assurance. Its main purpose is checking if the software is compliant with the user's expectations (project specification)

Types of tests

- Unit tests
- Integration tests
- System tests
- Acceptance tests

FIRST rule

- Fast
- Isolated/Independent
- Repeatable
- Self-checking
- Thorough

Software Testing – Fundamentals: JUnit 5

JUnit 5

Tool used for creating recurrent unit tests of sotware written in Java.

Installation

In order to add JUnit using Maven, you have to add:

```
<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-
jupiter-api -->
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.7.0-M1</version>
    <scope>test</scope>
</dependency>
```

References

Reference	Description
@Test	Test method
@BeforeEach	Method executed before each testing method
@AfterEach	Method executed after each testing method
@BeforeAll	Method executed once, at the beginning of tests in given class
@AfterAll	Method executed once, after execution of tests in given class
@DisplayName	Custom name of of testing method
@Disabled	Describes a method or class which should not be executed during tests

@Disabled	Describes a method or class, which should not be executed during tests

@ Test

```
class ExampleTest {

    @Test
    void firstTest() {

    }
}
```

@BeforeEach

```
class ExampleTest {

    @BeforeEach
    void init() {
        System.out.println("This message will be displayed before
execution of each testing method");
    }

    @Test
    void firstTest() {
        System.out.println("Executing first test...");
    }

    @Test
    void secondTest() {
        System.out.println("Executing second test...");
    }
}
```

@AfterEach

@AfterEach

```
class ExampleTest {

    @AfterEach
    void init() {
        System.out.println("This message will be displayed after
execution of each testing method");
    }

    @Test
    void firstTest() {
        System.out.println("Executing first test...");
    }

    @Test
    void secondTest() {
        System.out.println("Executing second test...");
    }
}
```

@BeforeAll

```
class ExampleTest {

    @BeforeAll
    void init() {
        System.out.println("This message will be displayed  once
before execution of all testing methods");
    }

    @Test
    void firstTest() {
        System.out.println("Executing first test...");
    }

    @Test
    void secondTest() {
```

```
        System.out.println("Executing second test...");
    }
}
```

@AfterAll

```
class ExampleTest {

    @AfterAll
    void init() {
        System.out.println("This message will be displayed  once after
execution of all testing methods);
    }

    @Test
    void firstTest() {
        System.out.println("Executing first test...");
    }

    @Test
    void secondTest() {
        System.out.println("Executing second test...");
    }
}
```

@DisplayName

```
class ExampleTest {

    @Test
    @DispalyName("First testing method")
    void firstTest() {

    }
}
```

@Disabled

```
class ExampleTest {

    @Test
    @Disabled
    void firstTest() {
        // code of this testing method will not be executed
    }
}
```

Given – When – Then

A poopular style of writing unit tests is the Given – When – Then approac. Thanks to that, tests are more systematized and more readable. Many developers also add comments separating the described sections

```
@Test
public void testAdd() {
    // Given
    Calculator calculator = new Calculator();

    // When
    int result = calculator.add(2, 3);

    // Then
    Assertions.assertEquals(5, result);
}
```

Assertions

Assertion is an instruction, which main purpose is to confirm, that the predicate is true in the given

Assertion is an instruction, which main purpose is to confirm, that the predicate is true in the given place of the code. Assertions' purpose is to check the correctness of test execution. They are static methods of class *org.junit.jupiter.api*, which allow to compare the actual results of certain method with the expected ones.

Assertion	Description
assertEquals(expected, actual)	Comparison of two values
assertNotNull(object)	Checking if the object has been initialized
assertNull(object)	Checking if the object has not been initialized
assertTrue(value)	Checking if the field or method returns value true
assertFalse(value)	Checking if the field or method returns value false
assertArrayEquals(expected, resultArray)	Comparison of two tables

assertEquals(expected, actual)

```
@Test
public void testEquals() {
    Assertions.assertEquals(5, Calculator.add(3, 2));
    Assertions.assertEquals(5, Calculator.add(3, 2), "Testing using
assertEquals");
}
```

assertNotNull(object)

```
@Test
public void testNotNull() {
    String text = "SDA Academy";
    Assertions.assertNotNull(text);
    Assertions.assertNotNull(text, "Testing using assertNotNull");
```

```
}
```

assertNull(object)

```
@Test
public void testNull() {
    String text = null;
    Assertions.assertNotNull(text);
    Assertions.assertNotNull(text, "Testing using assertNull");
}
```

assertTrue(value)

```
@Test
public void testTrue() {
    boolean value = true;
    Assertions.assertTrue(value);
    Assertions.assertTrue(value, "Testing using assertTrue");
}
```

assertFalse(value)

```
@Test
public void testFalse() {
    boolean value = false;
    Assertions.assertFalse(value);
    Assertions.assertFalse(value, "Testing using assertFalse");
}
```

assertArrayEquals(expected, resultArray)

```
@Test
public void testArrayEquals() {
    Assertions.assertArrayEquals(new int[]{1,2,3}, new int[]{1,2,3});
    Assertions.assertArrayEquals(new int[]{1,2,3}, new int[]{1,2,3},
"Testing using assertArrayEquals");
}
```

Software Testing – Fundamentals: AssertJ

AssertJ

Library, which main purpose is to improve readibility and maintenance of tests in Java. It increases readibility and in many cases it cares for conciseness of code.

Installation

In order to use AssertJ, you have to use the following dependency.

```
<!-- https://mvnrepository.com/artifact/org.assertj/assertj-core -->
<dependency>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-core</artifactId>
    <version>3.16.1</version>
    <scope>test</scope>
</dependency>
```

Syntax

In AssertJ we are using static methods from package *org.assertj.core.api*.

```
@Test
void shouldAddTwoNumbers() {
    int result = 1 + 3;
    Assertions.assertThat(result).isEqualTo(4);
}
```

Chain syntax

Thanks to chain syntax we can check few conditions at once.

```
@Test
void shouldAddTwoNumbers() {
    int result = 1 + 3;

    Assertions.assertThat(result)
        .isEqualTo(4)
        .isNotEqualTo(5)
        .isLessThan(6)
        .isGreaterThan(3)
        .isBetween(0, 10);
}
```

Software Testing – Fundamentals: TDD

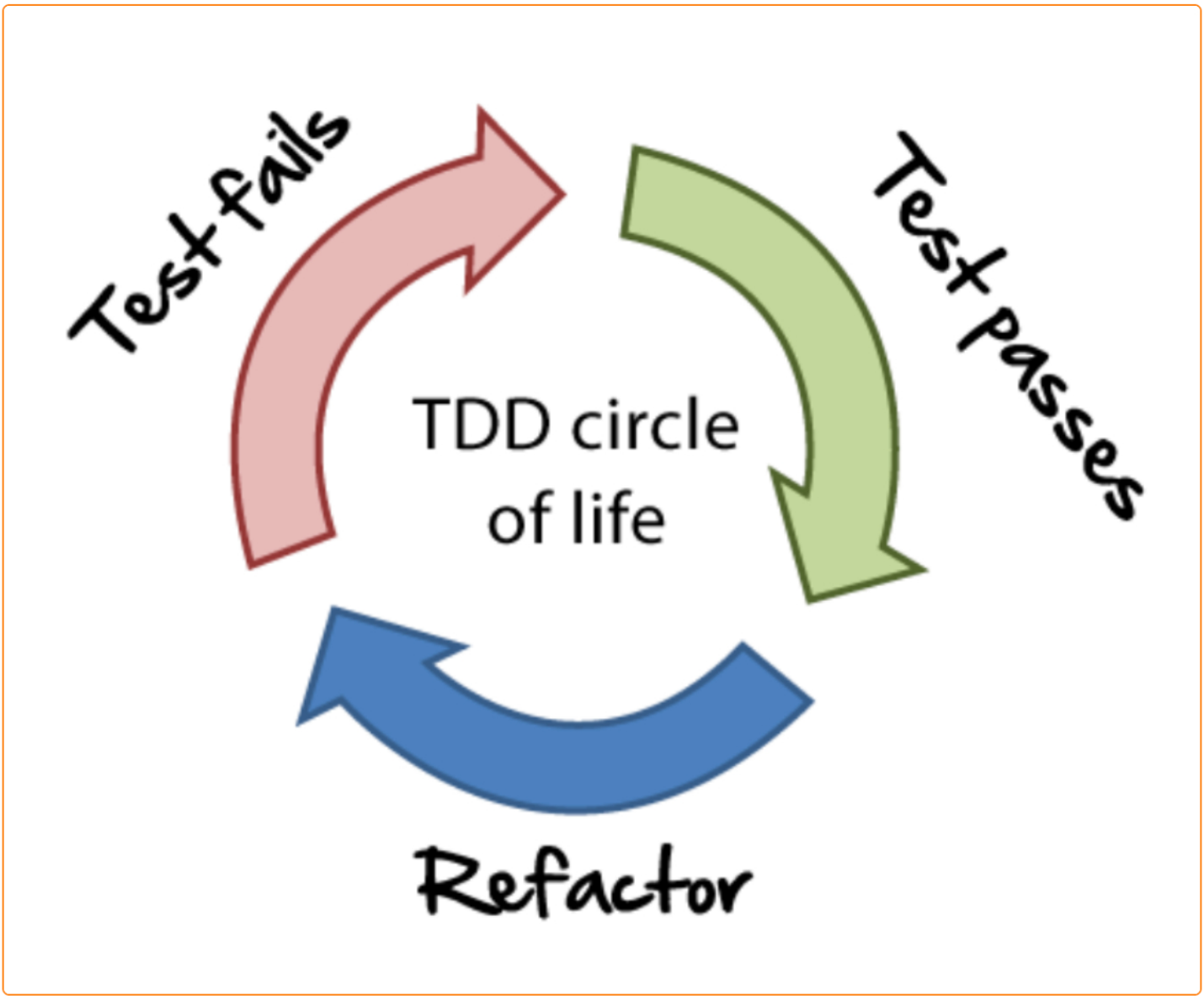
TDD

TDD

Test-driven development – a software development technique. It means multiple repeating few steps

1. First, a developer writes an automated test checking the added functionality. At this moment, the test should fail.
2. Then the development of the functionality happens. At this moment, the previously written test should pass.
3. In the last step, the developer refactors the written code in order to meet certain standards.

Red – Green – Refactor



Example

```
public class Fibonacci {
    public static int getValue(int element) {
        return null;
    }
}
```

Red – example

```
class FibonacciTest {

    @Test
    void shouldReturnZeroForZeroElement() {
        Assertions.assertEquals(0, Fibonacci.getValue(0));
    }

    @Test
    void shouldReturnOneForFirstElement() {
        Assertions.assertEquals(1, Fibonacci.getValue(1));
    }

    @Test
    void shouldReturnValue() {
        Assertions.assertEquals(8, Fibonacci.getValue(6));
        Assertions.assertEquals(144, Fibonacci.getValue(12));
        Assertions.assertEquals(987, Fibonacci.getValue(16));
    }
}
```

Green – example

```
public class Fibonacci {
    public static int getValue(int element) {
        if (element == 0) {
```

```
            return 0;
        }

        if (element == 1) {
            return 1;
        }

        int result;
        int position1;
        int position2;

        if (element > 1) {
            for (int i = 2; i <= element; i++) {
                result = position1 + position2;
                position2 = position1;
                position1 = result;
            }
        }

        return result;
    }
}
```

Refactor – example

```
public class Fibonacci {
    public static int getValue(int element) {
        if (element == 0 || element == 1) {
            return element;
        }

        return getValue(element - 2) + getValue(element - 1);
    }
}
```

Software testing – advanced: testing exceptions

---

assertThrows

JUnit 5 provides a set of methods for testing exceptions. We include:

- `assertThrows (Class<T> expectedType, Executable executable)`
- `assertThrows (Class<T> expectedType, Executable executable, String message)`
- `assertThrows (Class<T> expectedType, Executable executable, Supplier <String> messageSupplier)`

---

assertThrows

All the `assertThrows` methods allow you to pass the tested interaction using the `Executable` functional interface. However, the result can be verified at the level of the assertion being made, or later at the level of retrieving the exception object.

Example

```
@Test
void assert_throw_illegal_argument_exception_when_divide_by_0() {
    assertThrows(IllegalArgumentException.class, () -> {
        divide(10, 0);
    }, "divider can't be null");
}
```

Software testing – advanced: parametrized tests

Parameterized tests

With the help of parametrized tests, it is possible to perform a single test multiple times. What is more it can be achived with a different set of input data.

Parameterized tests JUnit 5

In order to use parameterized tests within the JUnit 5 framework, the following dependency should be used

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.6.1</version><!-- the current version may change -->
    <scope>test</scope>
</dependency>
```

@ParameterizedTest

This annotation is used to define parameterized tests in test classes

```
@ParameterizedTest
@ValueSource(strings = { "Testing", "JUnit", "SDA" })
void verify_if_value_source_is_not_null(String word) {
    assertNotNull(word);
}
```

@ValueSource

This annotation allows you to define the type of the input

Modifier and Type	Optional Element	Description
`byte[]`	`bytes`	The `byte` values to use as sources of arguments; must not be empty.
`char[]`	`chars`	The `char` values to use as sources of arguments; must not be empty.
`Class<?>[]`	`classes`	The `Class` values to use as sources of arguments; must not be empty.
`double[]`	`doubles`	The `double` values to use as sources of arguments; must not be empty.
`float[]`	`floats`	The `float` values to use as sources of arguments; must not be empty.
`int[]`	`ints`	The `int` values to use as sources of arguments; must not be empty.
`long[]`	`longs`	The `long` values to use as sources of arguments; must not be empty.
`short[]`	`shorts`	The `short` values to use as sources of arguments; must not be empty.
`String[]`	`strings`	The `String` values to use as sources of arguments; must not be empty.

@NullSource and @EmptySource

These annotations allow you to pass single values that are null and empty, respectively. The `NullAndEmptySource` annotation allows the use of both values as part of a defined parameterized set of tests.

@EnumSource

This annotation allows for passing parameterized objects of enumeration type:

```
    @ParameterizedTest
    @EnumSource(CurrencyConverter.class)
    void is_converted_money_higher_than_0(CurrencyConverter converter)
{
        assertTrue(converter.convertCurrency("USD", 20.0f)>0);
    }
```

@CsvSource

This annotation allows you to pass CSV literal parameterized tests:

```
public class Numbers {
    public static boolean isOdd(int input) {
        return input % 2 == 1;
    }
}
```

```
    @ParameterizedTest
    @CsvSource({"1,true", "2,false", "   4,false"})
    void isOdd_should_return_expected_result(String input, String
expected) {
        boolean actualResult = Numbers.isOdd(Integer.parseInt(input));
        assertEquals(Boolean.valueOf(expected), actualResult);
    }
```

@CsvFileSource

This annotation makes it possible to pass parameterized files in the CSV format to the tests:

```
public class Numbers {
    public static boolean isOdd(int input) {
        return input % 2 == 1;
    }
}
```

```
    @ParameterizedTest
    @CsvFileSource(resources = "/data.csv")
    void isOdd_should_return_expected_result(String input, String
expected) {
```

```
        boolean actualResult = Numbers.isOdd(Integer.parseInt(input));
        assertEquals(Boolean.valueOf(expected), actualResult);
    }
```

@MethodSource

This annotation allows you to pass methods that can provide custom objects so that parameterized tests can be more concrete.

@ArgumentSource

This annotation makes it possible to pass parameterized non-default data providers for testing.

Complete Lesson