Junit and Mockito

Contents

1. Introduction to testing

Software testing is the process of evaluating a software application to ensure it works as expected and meets user requirements.

Different types of testing:

1. **Unit testing**: Test individual components or functions in isolation to ensure they work as expected.
2. **Functional testing**: Assess whether the software meets specified functional requirements and behave as expected.
3. **Integration Testing**: Validate the interaction between integrated components or systems to detect interface defects.
4. **Smoke Testing**: A basic test to check basic functionalities of an application after a new build is deployed.
5. **Load Testing**: Tests the system's performance under expected load conditions to identify bottlenecks.
6. **Stress Testing:** Assess how system behave under extreme conditions beyond its operational capacity.
7. **Regression Testing:** Confirms new code changes do not adversely affect existing functionality.
8. **Sanity Testing:** A focused subset of regression testing to verify that specific functionality work after changes or bug fixes.
9. **Alpha testing:** Conducted internally by developers or testers to identify bugs before releasing the product to external users.
10. **Beta Testing:** Final testing phase conducted by real users in a real environment to gather feedback and identify any remaining issues.
11. **User Acceptance Testing (UAT):** Conducted by end-users to validate whether the system meets their needs and requirements.

Functional and non-functional testing -

Functional testing verifies that the software behaves according to the requirements, while non-functional testing evaluates aspects like performance, usability and reliability.

Test coverage: Test coverage measures the amount of code that is tested by unit tests, it helps to identify parts of codebase that lack testing.

Code smell: Code smell is a hint that something might be wrong in the code (eg duplicated code, long methods). Unit testing can help identify these issues by ensuring that tests fails when code changes introduce bugs.

TDD (Test Driven Development): is a software development approach in which test cases for each functionality are created and tested first and if test fails, the new code is written in order to pass the test and making code simple and bug-free. In simple terms, in TDD test is written first and then to pass the test, code is written accordingly.

Unit testing

Unit Testing is a software testing technique where individual units or components of a software application are tested in isolation to ensure they function as expected. A "unit" refers to the smallest testable part of an application, such as a method or function.

Why is Unit Testing Important?

- Detects issues early in the development lifecycle.
- Simplifies debugging by isolating problems.
- Facilitates code changes and refactoring without introducing new bugs.
- Provides documentation of how individual components are intended to work.

2. Junit

JUnit is a popular testing framework for Java that provides annotations and tools for writing unit tests. It allows developers to automate testing and integrates seamlessly with build tools like Maven and Gradle.

setting up Junit

1. **For Maven Projects**: Add the following dependency to the pom.xml file.

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.10.0</version>
    <scope>test</scope>
</dependency>
```

2. **For Gradle Projects**: Add the following to the dependencies block in the build.gradle file.

```
testImplementation 'org.junit.jupiter:junit-jupiter:5.10.0'
```

3. **For Standalone Setup**:
   Download the JUnit JAR files from the JUnit website and add them to your project's classpath.

First Test Case

Class to test

```
package org.example;

public class Calculator {
    public int multiply(int a, int b) {
        return a * b;
    }
}
```

Test class

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class CalculatorTest {
    @Test
    public void testMultiply(){
        Calculator calc = new Calculator();
        assertEquals(8,calc.multiply(2,4));
    }
}
```

A typical JUnit test case consists of the following components:

1. **Test Method**: A method annotated with @Test that contains the logic to test a specific function.
2. **Assertions**: Used to validate expected outcomes.
3. **Setup Code**: Initializes the environment or objects required for the test.
4. **Teardown Code**: Cleans up resources after a test (if needed).

Organizing test cases -

1. **Single Responsibility**: Each test method should test one specific functionality.
2. **Naming Conventions**:
    - Use descriptive names for test methods, e.g., testAdditionWithPositiveNumbers.
3. **Separation of Concerns**:
    - Separate the setup, execution, and validation logic for clarity.

Assertions

Assertions are used to validate the output of a test case. They ensure the tested code meets the expected criteria.

1. **assertEquals(expected, actual)**: Verifies if the actual value matches the expected value. With double values can have 3rd parameter delta, which defined amount of difference in decimal we can tolerate. For opposite case, we have assertNotEquals() too.

```
Assertions.assertEquals(10, result);
```

2. **assertNotNull(object)**: Validates that an object is not null. Similarly assertNull() also exists.

3. **assertThrows(exception, executable)**: Ensures an exception is thrown.

```
Assertions.assertThrows(IllegalArgumentException.class, () -> {
someMethodThatThrowsException();
});
```

4. **assertTrue(condition)**:Validates that a condition is true. Similarly assertFalse() exists.

5. **assertArrayEquals()**: When we want to verify value of array is equal to expected one.

Sample codes -

1. Using assetArrayEquals()

```
//Test case to verify Arrays.sort method
@Test
public void testSortArray() {
    //Working code
    int[] input = {5, 3, 1, 4, 2};
    Arrays.sort(input);

    int[] expected = {1, 2, 3, 4, 5};
    Assertions.assertArrayEquals(expected, input);
}
```

2. Palindrome testing

Main code

```
public class StringUtils {
    public boolean isPalindrome(String input) {
        String reversed = new StringBuilder(input).reverse().toString();
        return input.equals(reversed);
    }
}
```

Test class

```
public class StringUtilsTest {
    @Test
    public void testIsPalindrome() {
        StringUtils stringUtils = new StringUtils();
        Assertions.assertTrue(stringUtils.isPalindrome("madam"));
        Assertions.assertFalse(stringUtils.isPalindrome("hello"));
    }
}
```

Important Annotations

1. **@Disabled:**

- Temporarily disables a test.
- Useful for tests that are incomplete or failing.

```
@Disabled("Feature under development")
@Test
void incompleteTest() {
    Assertions.fail("Not implemented yet");
}
```

2. **@DisplayName**

- Customizes the display name of the test.

```
@DisplayName("Addition Test")
@Test
void testAddition() {
    Assertions.assertEquals(4, 2 + 2);
}
```

3. **@Tag**

- Tag tests for filtering during execution

```
@Tag("fast")
@Test
void fastTest() {
    Assertions.assertTrue(true);
}
```

Test Fixtures

Test fixtures are a set of methods used to set up the necessary environment and resources for testing and to clean them up afterward. They ensure that:

1. Each test starts in a known state.
2. Resources such as files, databases, or connections are properly handled.
3. Code duplication in test setup and teardown is minimized.

For example, if multiple tests rely on a database connection, a test fixture can establish the connection once and clean it up afterward.

Annotations in Test fixtures -

1. **@BeforeEach**

- Executes **before each test method**.
- Used to initialize or set up objects and data.
- Ensures every test starts in a clean state.

2. **@AfterEach**

- Executes **after each test method**.
- Used to clean up resources or reset configurations.

3. **@BeforeAll**

- Executes **once before all test methods** in the class.
- Used for time-consuming setup like database connections.
- Must be **static**.

4. **@AfterAll**

- Executes **once after all test methods** in the class.
- Used to release resources shared across tests.
- Must be **static**.

Code Example:

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

class CalculatorTest {
    private Calculator calculator;

    @BeforeAll
    static void initializeSuite() {
        System.out.println("Suite setup: Establishing shared resources.");
    }

    @BeforeEach
    void setup() {
        calculator = new Calculator();
        System.out.println("Test setup: Initializing calculator.");
    }

    @Test
    void testAddition() {
        System.out.println("Running testAddition.");
        Assertions.assertEquals(5, calculator.add(2, 3), "Addition test failed.");
    }

    @Test
    void testSubtraction() {
        System.out.println("Running testSubtraction.");
        Assertions.assertEquals(1, calculator.subtract(3, 2), "Subtraction test
failed.");
    }

    @AfterEach
    void tearDown() {
        System.out.println("Test teardown: Cleaning test-specific resources.");
    }

    @AfterAll
    static void finalizeSuite() {
        System.out.println("Suite teardown: Releasing shared resources.");
    }
}

class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    int subtract(int a, int b) {
        return a - b;
    }
}
```

Parameterized Test

Parameterized tests in JUnit allow you to run the same test logic repeatedly with different sets of input data. Instead of duplicating the same test multiple times with varying inputs, you define a single test method that accepts parameters.

1. @ParameterizedTest

- Marks a test method as parameterized.
- Works in conjunction with other data source annotations like @ValueSource, @CsvSource, etc.

2. @ValueSource

- Provides a fixed array of values to the test method.
- Supports primitive types (int, long, double), String, and Clas

```
class ParameterizedExample {
    @ParameterizedTest
    @ValueSource(ints = {1, 2, 3, 4, 5})
    void testIsEven(int number) {
        System.out.println("Testing with number: " + number);
        Assertions.assertTrue(number > 0, "Number should be positive");
    }


@ParameterizedTest
@ValueSource(strings = {"hello", "world", "junit"})
void testStringLength(String word) {
    Assertions.assertTrue(word.length() > 0, "String should not be empty");
}

}
```

3. @CsvSource

- Supplies a list of comma-separated values.
- Each line represents a set of inputs.

```
@ParameterizedTest
@CsvSource({
        "1, 1",
        "2, 4",
        "3, 9"
})
void testSquare(int number, int expectedSquare) {
    Assertions.assertEquals(expectedSquare, number * number, "Square calculation
failed");
}
```

4. Interview Questions

1. **Why BeforeAll and AfterAll methods need to be static?**

   @BeforeAll and @AfterAll methods are typically static because they are executed once at the class level, not per test instance. Since JUnit creates a new test class instance for each test method, static ensures these methods can run without needing a specific instance. In JUnit 5, non-static methods can be used if the class is annotated with @TestInstance(Lifecycle.PER_CLASS).

2. **Why is it recommended to keep test methods void and without parameters in JUnit?**

   Test methods in JUnit are kept void and without parameters because they are designed to check specific conditions or behaviors in isolation. A void return type ensures the method focuses on testing and not returning values, while no parameters make the tests simple and independent, running without needing extra inputs or setup during execution.