**JUnit Testing – Assignment Documentation**

**Exercise 1: Setting Up JUnit**

**Objective**

The purpose of this exercise is to set up the JUnit testing framework in a Java project environment. This involves configuring a Java project to support JUnit tests, integrating the necessary dependencies, and creating the initial test class to begin validating Java methods through unit testing.

**Scenario**

JUnit is a widely used testing framework for Java. This exercise focuses on preparing a development environment that supports test-driven development using JUnit.

**Steps to Perform**

1. Open an Integrated Development Environment (IDE) such as IntelliJ IDEA or Eclipse.
2. Create a new Java project with the necessary folder structure (typically src/main/java and src/test/java).
3. Set up Maven in the project by adding a pom.xml file at the root.
4. Add the JUnit dependency to the pom.xml file.

**Maven Dependency Configuration**

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

This dependency ensures that JUnit libraries are included only during the testing phase and are not packaged with production code.

**Creating the First Test Class**

Once the JUnit dependency is added and the project is built successfully, a test class can be created under the src/test/java directory.

**Sample Java Test Class**

import org.junit.Test;

import static org.junit.Assert.\*;

public class SampleTest {

@Test

public void testSum() {

int result = 2 + 3;

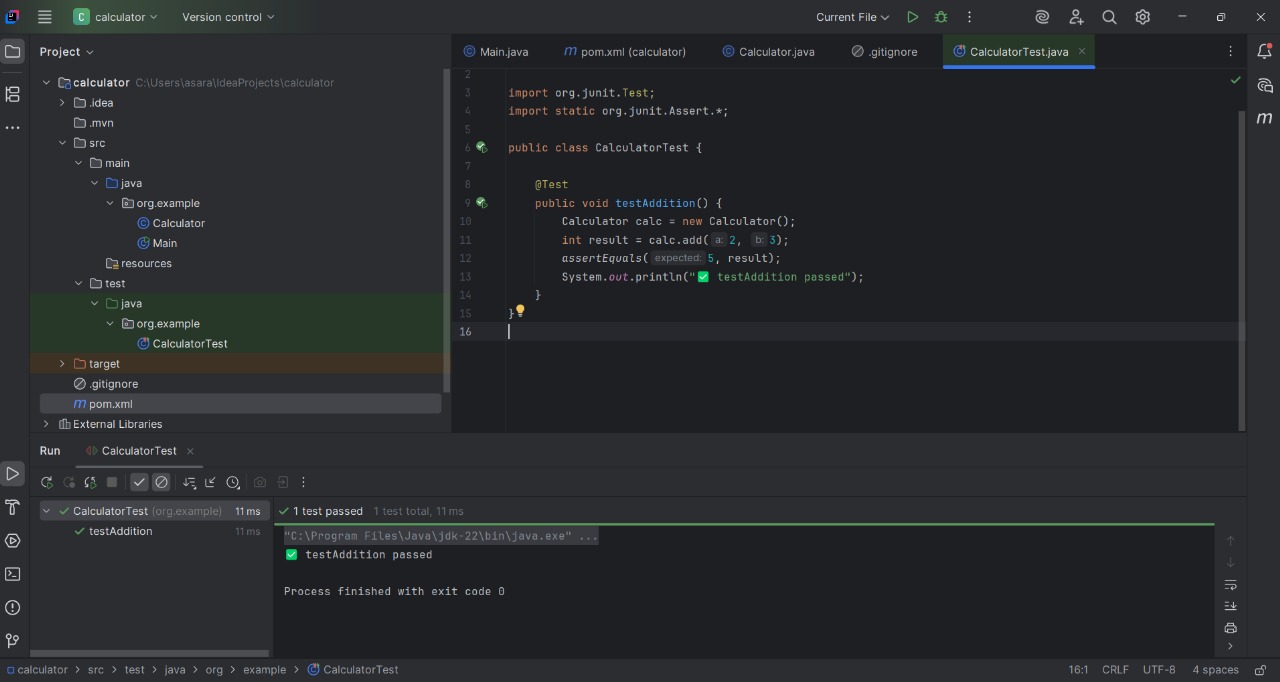
assertEquals(5, result);

}

}

This example demonstrates a basic test method annotated with @Test. It uses the assertEquals method to validate the sum of two integers.

**Output:**



**Exercise 3: Assertions in JUnit**

**Objective**

This exercise is aimed at validating the use of various JUnit assertions that are essential for checking test outcomes in unit testing. Assertions are core components in test-driven development and are used to compare expected results with actual outcomes in Java applications.

**Scenario**

The goal is to use multiple types of assertions provided by JUnit to verify logical conditions and data integrity during test execution. These assertions help identify whether the implemented business logic behaves as expected.

**Types of Assertions Demonstrated**

1. assertEquals – verifies that two values are equal
2. assertTrue – confirms that a condition evaluates to true
3. assertFalse – confirms that a condition evaluates to false
4. assertNull – checks if an object is null
5. assertNotNull – ensures that an object is not null

**Test Class Implementation**

import static org.junit.Assert.\*;

import org.junit.Test;

public class AssertionsTest {

@Test

public void testAssertions() {

assertEquals(5, 2 + 3);

assertTrue(5 > 3);

assertFalse(5 < 3);

assertNull(null);

assertNotNull(new Object());

}

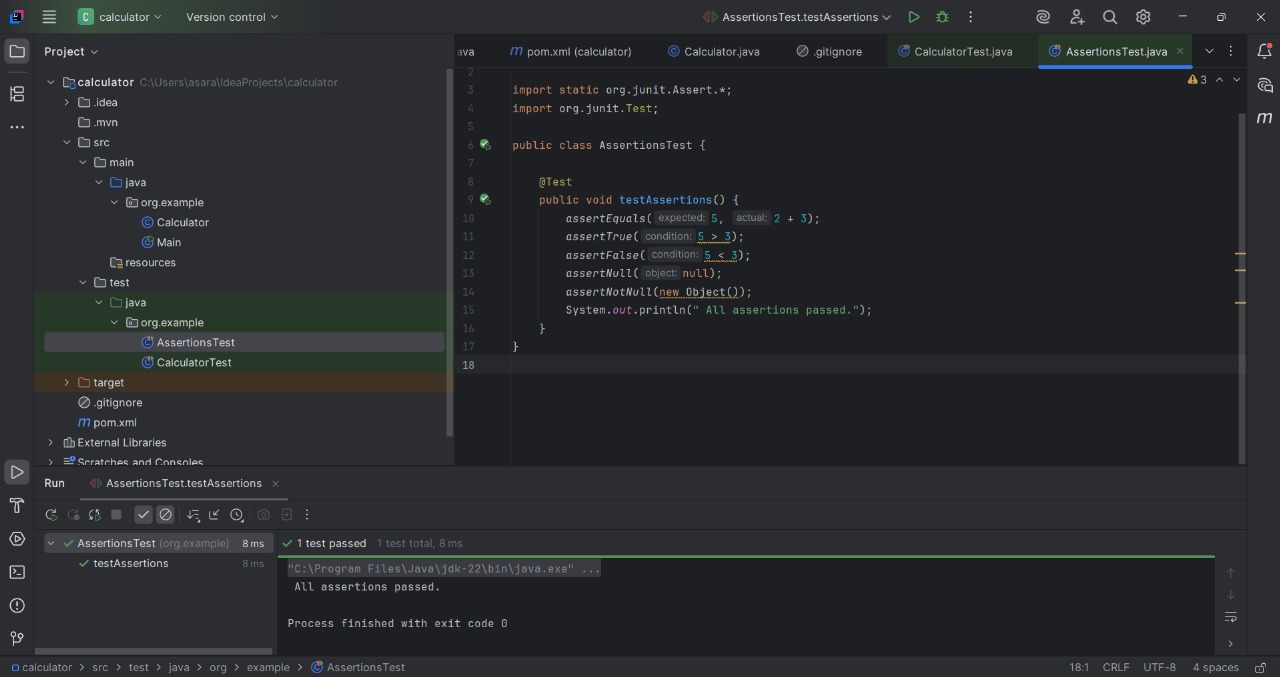
}

This test class includes a single method testAssertions() that demonstrates each of the assertions listed above using simple and straightforward logic.

**Execution Method**

To execute the test, right-click on the class file in the IDE and select the "Run" option, or use the Maven command mvn test if executing through the terminal. The JUnit runner will evaluate each assertion inside the test method.

**Output:**



**Exercise 4: Arrange-Act-Assert (AAA) Pattern, Test Fixtures, Setup and Teardown**

**Objective**

This exercise demonstrates the structured approach to writing unit tests using the Arrange-Act-Assert (AAA) pattern. It also introduces test fixtures and the use of setup and teardown methods in JUnit using the @Before and @After annotations. These features are essential for creating reusable, isolated, and predictable test cases in large-scale software systems.

**Scenario**

In software testing, structuring tests improves readability and maintainability. The AAA pattern provides a clean framework for organizing test code. The @Before annotation is used to define pre-test setup logic, while @After defines post-test cleanup logic. This approach avoids redundant code and ensures consistent test behavior.

**Arrange-Act-Assert Pattern**

Arrange: Initialize test objects and set up the conditions required for the test.  
Act: Call the method or logic under test.  
Assert: Verify that the outcome matches the expected result.

**Code Implementation – Calculator Test**

**Calculator.java (Business Logic)**

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int subtract(int a, int b) {

return a - b;

}

public int multiply(int a, int b) {

return a \* b;

}

public int divide(int a, int b) {

if (b == 0) {

throw new IllegalArgumentException("Division by zero is not allowed.");

}

return a / b;

}

}

**CalculatorTest.java (JUnit Test Class)**

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

private Calculator calculator;

@Before

public void setUp() {

calculator = new Calculator();

}

@After

public void tearDown() {

calculator = null;

}

@Test

public void testAddition() {

int result = calculator.add(10, 5);

assertEquals(15, result);

}

@Test

public void testSubtraction() {

int result = calculator.subtract(10, 3);

assertEquals(7, result);

}

@Test

public void testMultiplication() {

int result = calculator.multiply(4, 5);

assertEquals(20, result);

}

@Test

public void testDivision() {

int result = calculator.divide(20, 4);

assertEquals(5, result);

}

@Test(expected = IllegalArgumentException.class)

public void testDivisionByZero() {

calculator.divide(10, 0);

}

}

**Output:**

