**WEEK-2**

**JUnit Testing Exercises**

**Exercise 1: Setting Up JUnit Scenario:**

You need to set up JUnit in your Java project to start writing unit tests.

Steps:

1. Create a new Java project in your IDE (e.g., IntelliJ IDEA, Eclipse).

2. Add JUnit dependency to your project. If you are using Maven, add the following to your pom.xml:

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

3. Create a new test class in your project.

**Code:**

package com.example;

public class Calculate {

public int add(int x, int y) { return x+ y; }

public int subtract(int x, int y) { return x - y; }

public int multiply(int x, int y) { return x \* y; }

public int divide(int x, int y) {

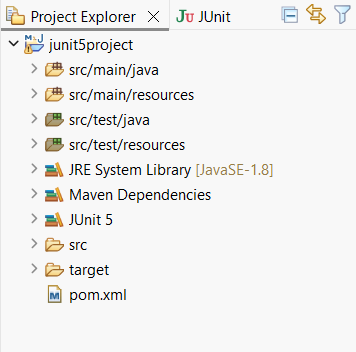
if (y == 0) throw new IllegalArgumentException("y cannot be zero");

return x / y;

}

}

**Output:**



**Explanation:**

* A Java testing framework called JUnit is used to verify that your code is operating as intended.
* I developed a Calculate class that included addition, subtraction, multiplication, and division.
* To enable the project to run test cases, I included the JUnit dependency in the pom.xml file.
* To verify each function with various inputs, I developed a distinct test class using @Test methods.
* Executing the tests demonstrates that the procedures function as planned, and JUnit makes it simple to identify problems like divide by zero.

**Exercise 3: Assertions in JUnit**

**Scenario:**

You need to use different assertions in JUnit to validate your test results.

Steps:

1. Write tests using various JUnit assertions.

Solution Code:

public class AssertionsTest (

@Test

public void testAssertions() {

// Assert equals assertEquals(5, 2+3):

// Assert true assertTrue(5>3);

// Assert false assertFalse(53);

// Assert null assertNull(null);

// Assert not null assertNotNull(new Object());

}

}

**Code:**

package com.example;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.\*;

public class AssertionsTest {

@Test

void testAssertions() {

assertEquals(9, 6-3);

assertTrue(5 >= 0);

assertFalse(7 < 4);

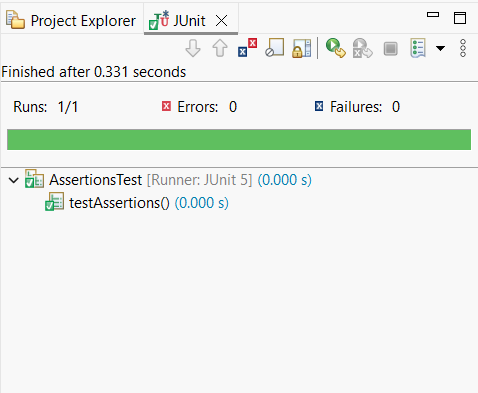
assertNull(null);

assertNotNull(new Object());

}

}

**Output:**

****

**Explanation:**

* To verify that my code was functioning properly, I utilized a variety of JUnit-provided assertion types in this task.
* assertEquals determines whether two values are equal, such as 6 + 3 = 9.
* assertTrue and assertFalse determine if a condition makes sense or not.
* While assertNotNull verifies that something exists (not null), assertNull guarantees that a value is null.
* These claims make test findings understandable and aid in identifying logical errors early in the development process.

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

**Scenario:**

You need to organize your tests using the Arrange-Act-Assert (AAA) pattern and use setup and teardown methods.

Steps:

1. Write tests using the AAA pattern.

2. Use @Before and @After annotations for setup and teardown methods.

**Code:**

package com.example;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculateTestAAA {

private Calculate calc;

@Before

public void setUp() {

calc = new Calculate();

System.out.println("Setup complete.");

}

@After

public void tearDown() {

calc = null;

System.out.println("Teardown complete.");

}

@Test

public void testAdd() {

int expected = 10;

int actual = calc.add(4, 6);

assertEquals(expected, actual);

}

@Test

public void testMultiply() {

int result = calc.multiply(8,6);

assertEquals(58, result);

}

@Test

public void testDivide() {

int result = calc.divide(9, 3);

assertEquals(3, result);

}

@Test

public void testSubtract() {

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

assertEquals(7, result);

}

@Test

public void testDivideByZero() {

try {

calc.divide(8, 0);

fail("Expected IllegalArgumentException");

} catch (IllegalArgumentException e) {

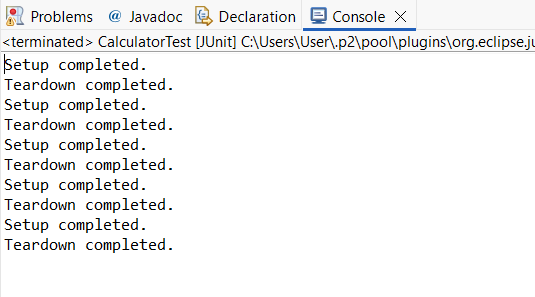
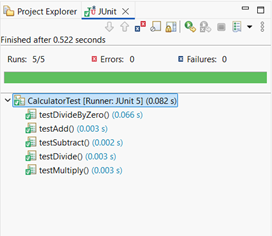
assertEquals("y cannot be zero", e.getMessage());

}

}

}

**Output:**

**Explanation:**

• The Arrange-Act-Assert (AAA) pattern is used in this exercise to arrange test methods in a clear and understandable manner.   
• Before every test, I created a Calculate object using the @Before annotation, which serves as the "Arrange" step.   
• The "Act" stage, where the primary operation is carried out, is the actual method call (such as add, multiply, etc.).   
• I verified the output using assertEquals and exception checks; this is the "Assert" step, where we make sure the outcome is what we expected.   
• To prevent residual data, the @After function clears the object after each test, ensuring that each one runs independently and fresh.