**Candidate Name:** SUCHITHRA M S  
**Superset ID:** 6430920

**WEEK – 2 HANDS ON EXERCISE (JAVA FSE DEEPSKILLING)**  
**Module 4 – Test Driven Development and Logging Framework (JUnit Testing)**

**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:
3. <dependency>
4. <groupId>junit</groupId>
5. <artifactId>junit</artifactId>
6. <version>4.13.2</version>
7. <scope>test</scope>
8. </dependency>

**Code Part (Calculation.java):**

package com.example.junitdemo;

public class Calculation {

public int multiply(int x, int y) {

return x \* y;

}

public int divide(int a, int b) {

if (b == 0) {

throw new IllegalArgumentException("Cannot divide by zero");

}

return a / b;

}

}

**Test Code (CalculationTest.java):**

package com.example.junitdemo;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class CalculationTest {

@Test

public void testMultiplication() {

Calculation calc = new Calculation();

int result = calc.multiply(4, 7);

System.out.println("Multiplication result: " + result);

assertEquals(28, result);

}

@Test

public void testDivision() {

Calculation calc = new Calculation();

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

System.out.println("Division result: " + result);

assertEquals(5, result);

}

@Test(expected = IllegalArgumentException.class)

public void testDivideByZero() {

Calculation calc = new Calculation();

calc.divide(10, 0); // Should throw exception

}

}

**Terminal Output For Test Pass Confirmation:**

A black screen with red numbers and white text

AI-generated content may be incorrect.

**Exercise 2: Assertions in JUnit**

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

**Solution Code (AssertionsTest.java):**

package com.example.junitdemo;

import static org.junit.Assert.\*;

import org.junit.Test;

public class AssertionsTest {

@Test

public void testAssertions() {

System.err.println("Starting testAssertions...");

assertEquals("Check if 2 + 3 equals 5", 5, 2 + 3);

System.err.println("Passed: 2 + 3 equals 5");

assertTrue("Check if 5 is greater than 3", 5 > 3);

System.err.println("Passed: 5 is greater than 3");

assertFalse("Check if 5 is not less than 3", 5 < 3);

System.err.println("Passed: 5 is not less than 3");

String sample = null;

assertNull("Expected sample to be null", sample);

System.err.println("Passed: sample is null");

String message = "JUnit is running";

assertNotNull("Expected message to be not null", message);

System.err.println("Passed: message is not null");

System.err.println("All assertions in testAssertions completed successfully.\n");

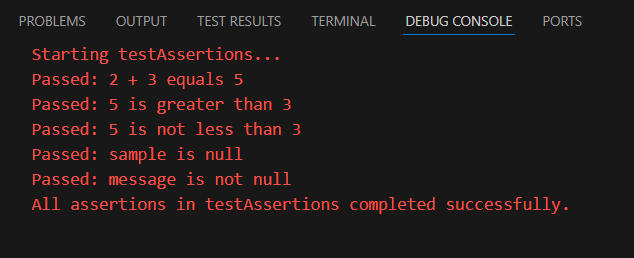
}

}

**Assertions in JUnit:**

| **Assertion** | **Purpose** |
| --- | --- |
| assertEquals | Validates if the expected value equals the actual value. |
| assertTrue | Confirms that a condition is true. |
| assertFalse | Confirms that a condition is false. |
| assertNull | Ensures the object is null. |
| assertNotNull | Ensures the object is not null. |

OUTPUT:



**Exercise 3: 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.

**Code Part (SimpleCalculation.java):**

package com.example.junitdemo;

public class SimpleCalculation {

public int addValues(int a, int b) {

return a + b;

}

public int subtractValues(int a, int b) {

return a - b;

}

}

**Test Code (SimpleCalculationTest.java):**

package com.example.junitdemo;

import org.junit.After;

import static org.junit.Assert.assertEquals;

import org.junit.Before;

import org.junit.Test;

public class SimpleCalculationTest {

private SimpleCalculation calc;

@Before

public void setUp() {

System.err.println("Setting up SimpleCalculation object...");

calc = new SimpleCalculation(); // Arrange

}

@After

public void tearDown() {

System.err.println("Cleaning up after test...\n");

calc = null;

}

@Test

public void testAddition() {

// Act

int result = calc.addValues(3, 2);

// Assert

assertEquals("Expected 3 + 2 to equal 5", 5, result);

System.err.println("Addition test passed: 3 + 2 = " + result);

}

@Test

public void testSubtraction() {

// Act

int result = calc.subtractValues(10, 4);

// Assert

assertEquals("Expected 10 - 4 to equal 6", 6, result);

System.err.println("Subtraction test passed: 10 - 4 = " + result);

}

}

OUTPUT:

A screenshot of a computer

AI-generated content may be incorrect.

**Mockito Exercises**

**Exercise 1: Mocking and Stubbing**

**Scenario:**  
You need to test a service that depends on an external API. Use Mockito to mock the external API and stub its methods.

**Solution Code (ClientHandlerTest.java):**

package com.example.junitdemo;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.mock;

import static org.mockito.Mockito.when;

public class ClientHandlerTest {

@Test

public void shouldReturnStubbedInfoFromRemoteService() {

// Arrange

RemoteService mockRemote = mock(RemoteService.class);

when(mockRemote.fetchInfo()).thenReturn("Simulated Response");

ClientHandler handler = new ClientHandler(mockRemote);

// Act

String output = handler.processRequest();

// Assert

assertEquals("Simulated Response", output);

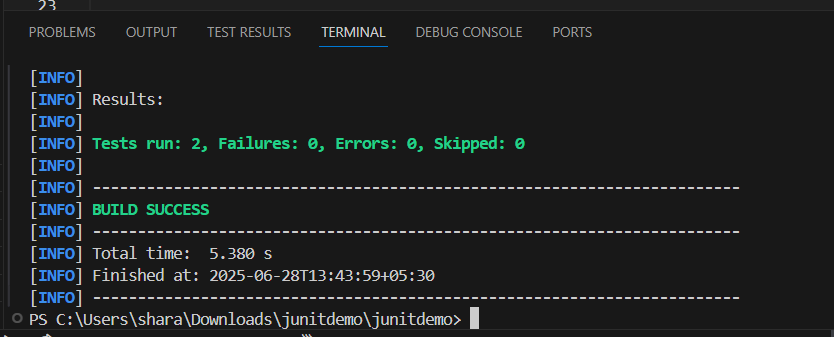
System.out.println("Test Passed — Received: " + output);

}

}

**Code Explanation:**

* **RemoteService.java:** Interface to simulate an external service.
* **ClientHandler.java:** Uses RemoteService to process requests.
* **ClientHandlerTest.java:** Mocks RemoteService and verifies the behavior.



**Exercise 2: Verifying Interactions**

**Scenario:**  
You need to ensure that a method is called with specific arguments.

**Solution Code (UserProcessorTest.java):**

package com.example.junitdemo;

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.mock;

import static org.mockito.Mockito.verify;

public class UserProcessorTest {

@Test

public void testInteractionWithApi() {

// Arrange

ApiClient mockApi = mock(ApiClient.class);

UserProcessor processor = new UserProcessor(mockApi);

// Act

processor.processUser();

// Assert/Verify

verify(mockApi).fetchUserDetails();

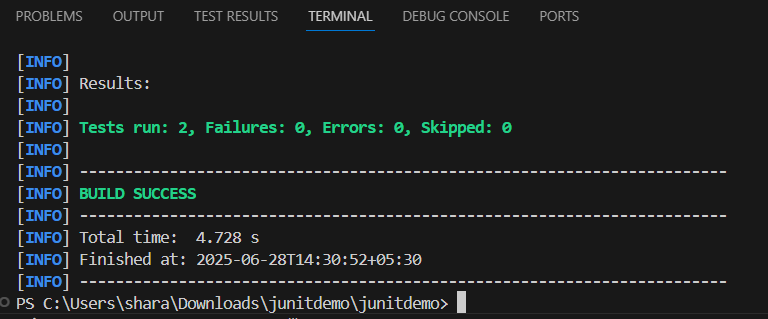
System.out.println("Verified: fetchUserDetails() was called.");

}

}

**Code Explanation:**

* **ApiClient.java:** Simulates an external API client.
* **UserProcessor.java:** Class that uses ApiClient.
* **UserProcessorTest.java:** Verifies that fetchUserDetails() is called during the test.



**SLF4J Logging Exercises**

**Exercise 1: Logging Error Messages and Warning Levels**

**Task:**  
Write a Java application that demonstrates logging error messages and warning levels using SLF4J.

**Step-by-Step Solution:**

1. Add SLF4J and Logback dependencies to your pom.xml:
2. <dependency>
3. <groupId>org.slf4j</groupId>
4. <artifactId>slf4j-api</artifactId>
5. <version>1.7.30</version>
6. </dependency>
7. <dependency>
8. <groupId>ch.qos.logback</groupId>
9. <artifactId>logback-classic</artifactId>
10. <version>1.2.3</version>
11. </dependency>
12. Create a Java class that uses SLF4J for logging:

**Code (AppLogger.java):**

package com.example.junitdemo;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class AppLogger {

private static final Logger log = LoggerFactory.getLogger(AppLogger.class);

public static void main(String[] args) {

log.error("Error: Database

A screenshot of a computer program

AI-generated content may be incorrect.