**Digital Nurture 4.0 Deep Skilling - Java FSE**  
**WEEK –2 Hands-on Exercises**  
**Module 4 – Test driven development and Logging framework**

1. **JUnit Testing Exercise**

**1.Exercise 1: Setting Up JUnit**

**Scenario:** You need to set up JUnit in your Java project to start writing unit tests

**Solution:**

**JUnit:**

JUnit is a widely used testing framework for the Java programming language. It helps developers write and run repeatable automated tests to ensure that individual units of code (typically methods in classes) work as expected.

It follows the principles of unit testing, which focuses on testing small parts of an application in isolation.

**Code:**

**Pom.xml**  
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example.junitdemo</groupId>

<artifactId>JUnitSetup1</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>JUnitSetup1</name>

<url>http://www.example.com</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

<dependencies>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>5.7.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-params</artifactId>

<version>5.7.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0-M5</version>

</plugin>

</plugins>

</build>

</project>

**App.java**

package com.example.junitdemo.JUnitSetup1;

public class App {

public int add(int a, int b) {

return a + b;

}

public boolean isPositive(int number) {

return number > 0;

}

public String greet(String name) {

if (name == null) {

return null;

}

return "Hello, " + name + "!";

}

}

**AppTest.java**

package com.example.junitdemo.JUnitSetup1;

import org.junit.jupiter.api.Test;

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

public class AppTest {

App app = new App();

@Test

public void testAddition() {

assertEquals(8, app.add(5, 3));

}

@Test

public void testIsPositiveTrue() {

assertTrue(app.isPositive(10));

}

@Test

public void testIsPositiveFalse() {

assertFalse(app.isPositive(-5));

}

@Test

public void testGreeting() {

assertEquals("Hello, Lokesh!", app.greet("Lokesh"));

}

@Test

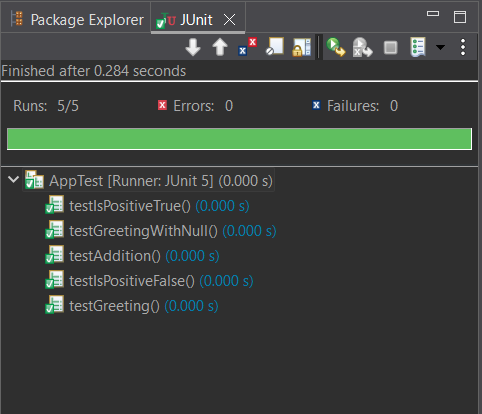
public void testGreetingWithNull() {

assertNull(app.greet(null));

}

}

**Output:**



**Explanation:**

1. Create a new Maven project.
2. Open pom.xml.
3. Add JUnit dependency:
4. Create a Java class with methods to be tested.
5. Create a new test class in src/test/java.
6. Annotate test methods with @Test.
7. Use assertion methods like assertEquals.
8. Run test

**2. Exercise 3: 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.assertions;

import org.junit.jupiter.api.Test;

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

public class AssertionsTest {

@Test

public void testAssertions() {

assertEquals(5, 2 + 3);

assertTrue(5 > 3);

assertFalse(5 < 3);

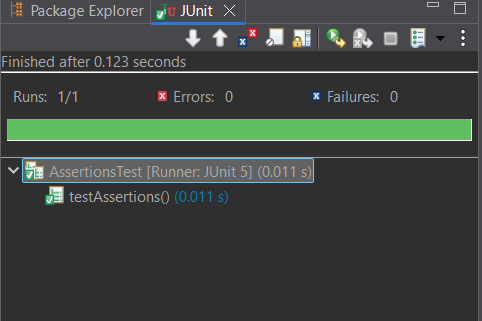
assertNull(null);

assertNotNull(new Object());

}

}

**Output:**



**Explanation:**

1. Create a test class named AssertionsTest.
2. Define a test method and annotate it with @Test.
3. Use assertEquals to verify that 2 + 3 equals 5.
4. Use assertTrue to check that the condition 5 > 3 is true.
5. Use assertFalse to ensure that the condition 5 < 3 is false.
6. Use assertNull to confirm that a value is null.
7. Use assertNotNull to confirm that a value is not null.

**3. 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.

**Solution:**

**Code:**

**Calculator.java**

package com.example.junitdemo.JUnitSetup1;

public class Calculator {

public int add(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("Cannot divide by zero");

return a / b;

}

}

**CalculatorTest.java**

package com.example.junitdemo.JUnitSetup1;

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

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

public class CalculatorTest {

private Calculator calculator;

@BeforeEach

public void setUp() {

calculator = new Calculator();

System.out.println(" Setup: Calculator instance created");

}

@AfterEach

public void tearDown() {

calculator = null;

System.out.println(" Teardown: Calculator instance destroyed");

}

@Test

public void testAdd() {

int a = 10, b = 5;

int result = calculator.add(a, b);

assertEquals(15, result);

System.out.println("testAdd passed");

}

@Test

public void testMultiply() {

int a = 4, b = 3;

int result = calculator.multiply(a, b);

assertEquals(12, result);

System.out.println(" testMultiply passed");

}

@Test

public void testDivideByZero() {

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

calculator.divide(10, 0);

});

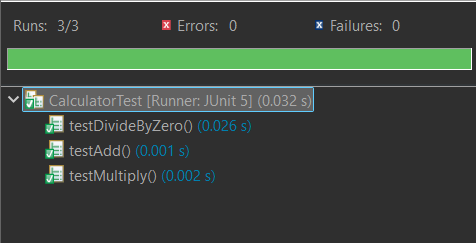
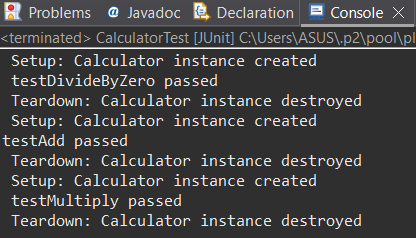
assertEquals("Cannot divide by zero", exception.getMessage());

System.out.println(" testDivideByZero passed");

}

}

**Output:**

**Explanation:**

1. Use the Arrange-Act-Assert (AAA) pattern to structure the test method:

Arrange: Set up test data and objects.

Act: Call the method being tested.

Assert: Verify the result using assertions.

1. Use @Before annotation to define a method that runs before each test to set up common test data or environment.
2. Use @After annotation to define a method that runs after each test to clean up or reset resources.
3. **Mockito Hands-On Exercises**

**1. 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:**

**Pom.xml**

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>mockito-demo</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>mockito-demo</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter</artifactId>

<version>5.9.3</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>5.2.0</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0-M5</version>

</plugin>

</plugins>

</build>

</project>

**ExternalApi.java**

package com.example.mockito\_demo;

public interface ExternalApi {

String getData();

}

**MyService.java**

package com.example.mockito\_demo;

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

**MyServiceTest.java**

package com.example.mockito\_demo;

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.\*;

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

public class MyServiceTest {

@Test

public void testFetchDataReturnsMockValue() {

ExternalApi mockApi = mock(ExternalApi.class);

when(mockApi.getData()).thenReturn("Mock Data");

MyService service = new MyService(mockApi);

String result = service.fetchData();

assertEquals("Mock Data", result);

System.out.println("fetchData() returned mocked value correctly");

}

@Test

public void testVerifyInteraction() {

ExternalApi mockApi = mock(ExternalApi.class);

MyService service = new MyService(mockApi);

service.fetchData();

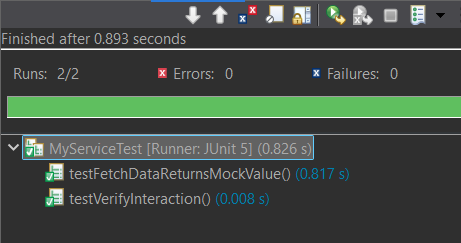
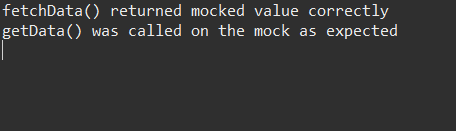
verify(mockApi).getData();

System.out.println("getData() was called on the mock as expected");

}

}

**Output:**

**Explanation:**

1. Create a mock object of the ExternalApi using Mockito.mock().
2. Stub the getData() method to return "Mock Data" using when(...).thenReturn(...).
3. Inject the mock into the MyService constructor.
4. Call the method fetchData() on MyService.
5. Use assertEquals to verify that the returned result matches the stubbed value "Mock Data".

**2. Exercise 2: Verifying Interactions**

**Scenario:**

You need to ensure that a method is called with specific arguments

**Solution:**

**Code:**

**MyServiceTest.java**

package com.example.mockito\_demo;

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.\*;

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

public class MyServiceTest {

@Test

public void testFetchDataReturnsMockValue() {

ExternalApi mockApi = mock(ExternalApi.class);

when(mockApi.getData()).thenReturn("Mock Data");

MyService service = new MyService(mockApi);

String result = service.fetchData();

assertEquals("Mock Data", result);

System.out.println("fetchData() returned mocked value correctly");

}

@Test

public void testVerifyInteraction() {

ExternalApi mockApi = mock(ExternalApi.class);

MyService service = new MyService(mockApi);

service.fetchData();

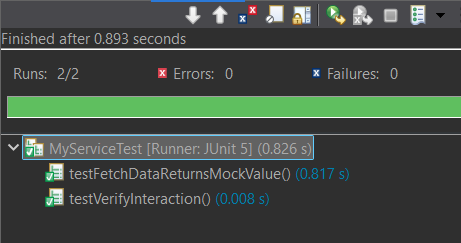
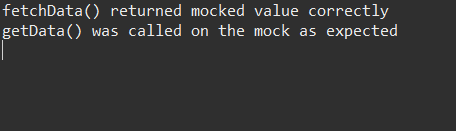
verify(mockApi).getData();

System.out.println("getData() was called on the mock as expected");

}

}

**Output:**

**Explanation:**

1. Create a mock object for ExternalApi using Mockito.mock().
2. Pass the mock to the MyService constructor.
3. Call the fetchData() method on the MyService instance.
4. Use verify() to check that getData() was called on the mock object.