Module-4 Test driven development and Logging framework

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.

**Step 1:** Create a new Java project using your IDE (e.g., IntelliJ IDEA or Eclipse).  
**Project Name:** JUnitSetupDemo

**Step 2:** Add JUnit dependency using Maven. In the pom.xml file, include:

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

</dependencies>

**Step 3:** Create a test class CalculatorTest.java under src/test/java:

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

@Test

public void testAddition() {

int result = 2 + 3;

assertEquals(5, result);

}

}

Output:

-------------------------------------------------------

T E S T S

-------------------------------------------------------

Running CalculatorTest

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.005 sec

Results :

Tests run: 1, Failures: 0, Errors: 0

BUILD SUCCESS

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(5 < 3);

// Assert null

assertNull(null);

// Assert not null

assertNotNull(new Object());

}

}

import org.junit.Test;

import static org.junit.Assert.\*;

public class AssertionsTest {

@Test

public void testVariousAssertions() {

// Assert equality

assertEquals("Sum should be 5", 5, 2 + 3);

// Assert inequality

assertNotEquals("Values should not be equal", 10, 2 \* 3);

// Assert true condition

assertTrue("Condition should be true", 10 > 5);

// Assert false condition

assertFalse("Condition should be false", 3 > 10);

// Assert null

Object obj1 = null;

assertNull("Object should be null", obj1);

// Assert array equality

int[] expected = {1, 2, 3};

int[] actual = {1, 2, 3};

assertArrayEquals("Arrays should be equal", expected, actual);

}

}

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.

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

private Calculator calculator;

// Setup method - runs before each test

@Before

public void setUp() {

calculator = new Calculator();

System.out.println("Setup: Calculator initialized");

}

// Teardown method - runs after each test

@After

public void tearDown() {

calculator = null;

System.out.println("Teardown: Calculator cleared");

}

// Test using Arrange-Act-Assert (AAA) pattern

@Test

public void testAddition() {

// Arrange

int a = 5;

int b = 3;

// Act

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

// Assert

assertEquals(8, result);

}

@Test

public void testSubtraction() {

// Arrange

int a = 10;

int b = 4;

// Act

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

// Assert

assertEquals(6, result);

}

}

Sample Calculator class for reference:

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int subtract(int a, int b) {

return a - b;

}

}

Output:  
Setup: Calculator initialized

Teardown: Calculator cleared

Setup: Calculator initialized

Teardown: Calculator cleared

-------------------------------------------------------

T E S T S

-------------------------------------------------------

Running CalculatorTest

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.006 sec

Results :

Tests run: 2, Failures: 0, Errors: 0

BUILD SUCCESS

Mockito  
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.

Steps:

1. Create a mock object for the external API.

2. Stub the methods to return predefined values.

3. Write a test case that uses the mock object.

Solution Code:

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

public class MyServiceTest {

@Test

public void testExternalApi() {

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

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

MyService service = new MyService(mockApi);

String result = service.fetchData();

assertEquals("Mock Data", result);

}

}

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

interface ExternalApi {

String getData();

}

class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

public class MyServiceTest {

@Test

public void testExternalApi() {

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

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

MyService service = new MyService(mockApi);

String result = service.fetchData();

assertEquals("Mock Data", result);

}

}

Exercise 2: Verifying Interactions

Scenario:

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

Steps:

1. Create a mock object.

2. Call the method with specific arguments.

3. Verify the interaction.

Solution Code:

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

MyService service = new MyService(mockApi);

service.fetchData();

verify(mockApi).getData();

}

}

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

public class MyServiceTestVerify {

@Test

public void testVerifyInteraction() {

ExternalApi mockApi = mock(ExternalApi.class);

MyService service = new MyService(mockApi);

service.fetchData();

verify(mockApi).getData(); // ensures getData() was called

}

}

SLF4J

Exercise 1: Logging Error Messages and Warning Levels with SLF4J

Step 1: Add Dependencies in pom.xml

<dependencies>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.30</version>

</dependency>

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.2.3</version>

</dependency>

</dependencies>

Step 2: Java Code for Logging

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class LoggingExample {

private static final Logger logger = LoggerFactory.getLogger(LoggingExample.class);

public static void main(String[] args) {

logger.error("This is an error message");

logger.warn("This is a warning message");

}

}

Output:

16:30:12.345 [main] ERROR LoggingExample - This is an error message

16:30:12.346 [main] WARN LoggingExample - This is a warning message