**Digital Nurture 4.0 – Week 2**

**ii).JUNIT TESTING EXERCISES**

**1. JUnit\_Basic Testing Exercises:**

**Exercise 1: Setting Up Junit**

**Program:**

This JUnit test class checks simple calculator operations. The test Addition method verifies 10 + 5 equals 15, printing helpful messages. The test Division By Zero method checks that dividing by zero correctly throws an Arithmetic Exception.

**CODE:**

package org.example;

import org.junit.jupiter.api.Test;

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

public class CalculatorTest {

@Test

void testAddition() {

int result = 10 + 5;

System.out.println("Running testAddition...");

System.out.println("Result of 10 + 5 is: " + result);

assertEquals(15, result);

System.out.println("testAddition passed\n");

}

@Test

void testDivisionByZero() {

System.out.println("Running testDivisionByZero...");

assertThrows(ArithmeticException.class, () -> {

int x = 1 / 0;

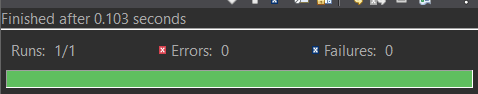
});

System.out.println("testDivisionByZero passed \n");

}

}

**Output:**

****

**Exercise 3: Assertions in JUnit**

**Program:**

This JUnit test class demonstrates the use of various assertions to verify expected outcomes in unit testing. It checks equality, boolean conditions, and nullability to ensure the program behaves correctly. These assertions help detect errors automatically during testing.

**CODE:**

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

import org.junit.jupiter.api.Test;

public class AssertionsTest {

@Test

public void testAssertions() {

assertEquals(5, 2 + 3);

assertTrue(5 > 3);

assertFalse(5 < 3);

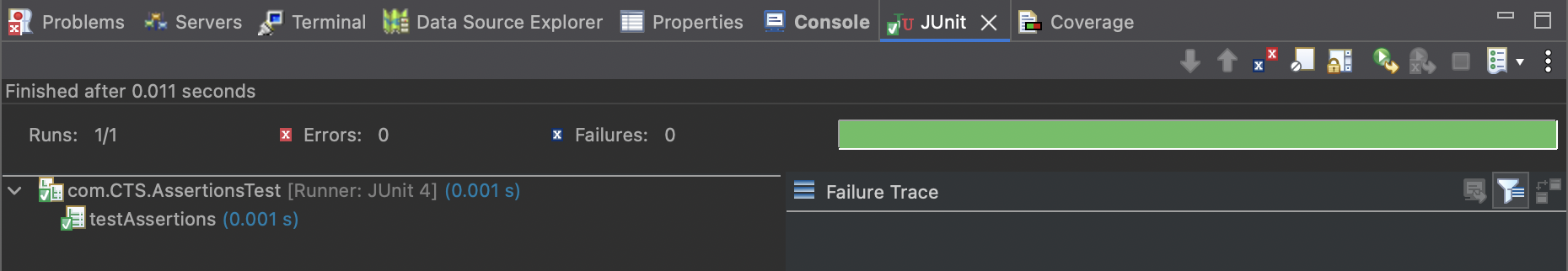
assertNull(null);

assertNotNull(new Object());

}

}

**Output:**



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

**Program:**

This program demonstrates the Arrange-Act-Assert (AAA) testing pattern in JUnit, where test data is arranged, the action is performed, and then results are asserted. It also uses @Before for setup and @After for cleanup after each test, ensuring test isolation. This pattern improves the readability and maintainability of tests.

**CODE:**

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

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

public class CalculatorTest {

private Calculator calculator;

@BeforeEach

public void setUp() {

calculator = new Calculator();

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

}

@AfterEach

public void tearDown() {

calculator = null;

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

}

@Test

public void testAddition() {

int a = 5;

int b = 3;

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

assertEquals(8, result);

}

@Test

public void testSubtraction() {

int a = 10;

int b = 4;

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

assertEquals(6, result);

}

}

class Calculator {

public int add(int x, int y) {

return x + y;

}

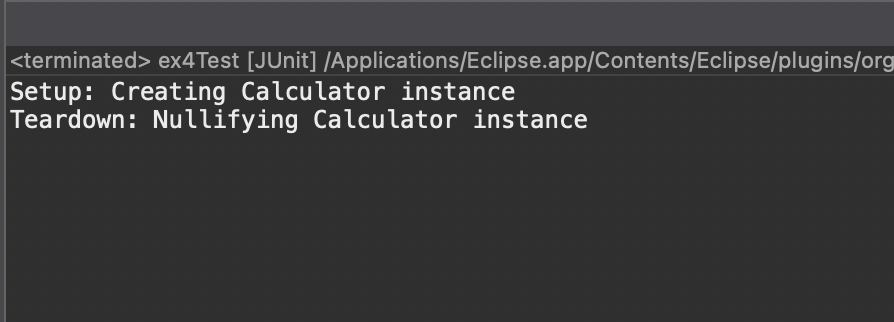
public int subtract(int x, int y) {

return x - y;

}

}

**Output:**



**3.Mockito Hands-On Exercises**

**Exercise 1: Mocking and Stubbing**

**Program:**

This code uses Mockito to create a *mock* of the ExternalApi, stubbing its getData() method to return a fixed string "Mock Data". Then it injects this mock into MyService and calls fetchData(), which should internally use the mocked API. Finally, it verifies the returned value is the stubbed "Mock Data".

**CODE:**

***1. ExternalApi.java***

public interface ExternalApi {

String getData();

}

***2. MyService.java***

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

***3.MyServiceTest.java***

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

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

public class MyServiceTest {

@Test

public void testExternalApi() {

ExternalApi mockApi = mock(ExternalApi.class);

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

MyService service = new MyService(mockApi);

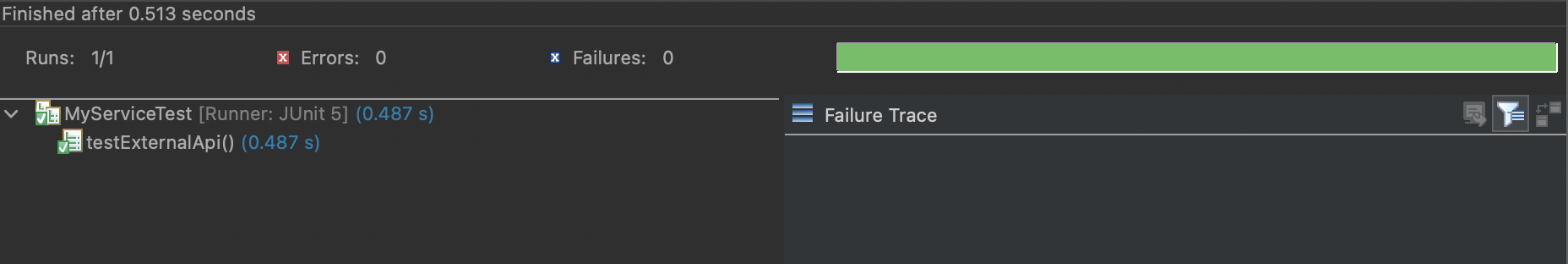
String result = service.fetchData();

assertEquals("Mock Data", result);

}

}

**Output:**



**Exercise 2: Verifying Interactions**

**Program:**

This code uses Mockito to create a mock ExternalApi and inject it into MyService.  
Then it calls service.fetchData(), which internally is expected to call mockApi.getData().  
Finally, verify(mockApi).getData() confirms that the getData() method was actually invoked.

**CODE:**

***1.ExternalApi.java***

public interface ExternalApi {

String getData();

}

***2.MyService.java***

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

***3.MyServiceTest.java***

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

ExternalApi mockApi = mock(ExternalApi.class);

MyService service = new MyService(mockApi);

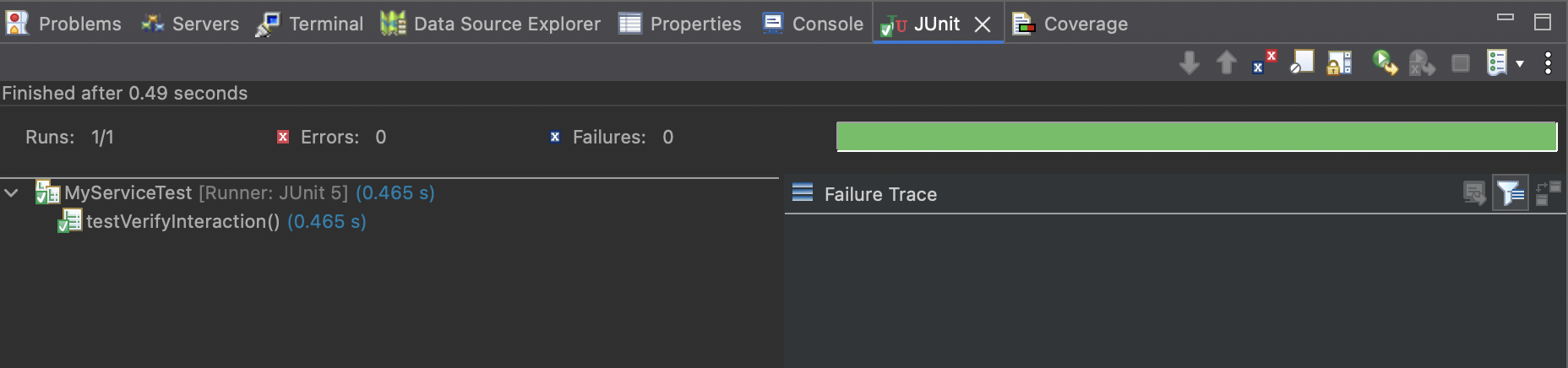
service.fetchData();

verify(mockApi).getData();

}

}

**Output:**



**6. SL4J Logging exercises**

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

*Step 1:*

Add the following dependencies on 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:*

In src/main/java/com/example create an Interface named LoggingExample.java.

LoggingExample.java:

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");

}

}

**Ouput:**

